University of Alberta

FORMALLY UNDERSTANDING THE BEHAVIOR OF A FRAMEWORK'S TRANSACTION MANAGEMENT

by

©

Nan Niu

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2003

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisisitons et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-87852-X
Our file Notre référence
ISBN: 0-612-87852-X

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

University of Alberta

Library Release Form

Name of Author: Nan Niu

Title of Thesis: Formally Understanding the Behavior of a Framework's Transaction Management

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Date: Sep 5, 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Formally Understanding the Behavior of a Framework's Transaction Management** submitted by Nan Niu in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. H. James Hoover
Supervisor

Dr. James Miller

Dr. Ken Wong

Date: Sep 2, 2003

Abstract

As the size and complexity of software systems increase, the overall system structure, which is also known as *software architecture*, becomes increasingly critical in software design and development. *Formal methods* provide mathematically based techniques that are particularly useful for systematically specifying and verifying computer systems. This thesis is a case study on leveraging formal methods to Prothos, a domain-specific database-centric web application framework. It is motivated in light of the question — **whether and how formalism can help software architecture**. Both structual and behavioral properties of the transaction management component in the framework are studied. Formal approaches based on model checking techniques are presented. Our practice verifies the current implementation, clarifies implicit assumptions, and provides potential improvements for the software framework. Also, our work can help develop a systematic way to tackle an architectural study.

To my mother and father with love and gratitude.

Acknowledgements

I have been lucky to have had opportunities to work with many brilliant people during my course study and thesis research. This has definitely helped to shape my thesis in ways I would never have been able to achieve on my own.

First and foremost, I would like to thank my supervisor, Jim Hoover, not only for his scholarly guidance and keen editorial eyes, but also for his encouragement and support throughout the work. His patience of going into gory details, his tolerance of my wayward requests, and above all, his willingness to contribute insightful comments to my work, have all been instrumental in making this thesis a reality.

This work could not have been done without the generous assistance of James Konkin at Avra Software. His authoritative, detailed, and prompt responses have solved many puzzles about Prothos. I am especially grateful to Matt Edwards for informative conversations and help about Prothos.

I wish to express my sincere gratitude to my committee: Dr. James Miller and Dr. Ken Wong. The final version of this thesis has greatly benefited from their comments. I owe a great debt of gratitude to Dr. Miller whose expertise and help in software verification and model evolution has been a valuable contribution to my work.

Thanks also go to SERL members: Dr. Eleni Stroulia, Dr. Paul Sorenson, Daqing Hou, Changyu Yin, Julian Fogel, Pablo Figueroa, and John Shillington for their discussion with respect to my work. Changyu proofreaded the early version of the thesis. Her comments greatly improved my presentation and eleminated a number of writing errors.

I also appreciate the advice of Dr. Steve Easterbrook, Dr. Amy Felty, Dr. Osmar Zaiane, Dr. Joerg Sander, Dr. Davood Rafiei, and Dr. Mario Nascimento. Many thanks to the Department of Computing Science and Jim for offering me the financial assistantship during my study.

Last, but certainly not least, I would like to thank my parents for their never-ending and unwavering love, care, support, and understanding which have clung to me all my life. It is on their behalf that this thesis is dedicated. I am similarly grateful to other members in our family: maternal grandmother, paternal grandmother, maternal uncle, maternal aunt, and my lovely maternal cousin. All my families have been waiting so long for this moment to come true; I am glad that their waiting has finally been rewarded. They are what makes it all worthwhile!

Contents

1 Introduction 1
1.1 Motivation 1
1.2 Thesis Structure 2

2 Background Practice and Related Research 4
2.1 DBMS 4
2.1.1 RDBMS 6
2.1.2 OODBMS 6
2.2 Software Framework 7
2.2.1 Software Architecture 9
2.2.2 Prothos 9
2.2.3 POM Structure 12
2.3 Formal Methods 14
2.3.1 Model Checking 15
2.4 Related Work 18
2.5 Thesis Scope 20

3 POM Architecture 21
3.1 Data Models 21
3.1.1 Object-Oriented Data Model 21
3.1.2 Relational Data Model 22
3.1.3 POM Data Model 24
3.2 Ontology 26
3.2.1 POM::Object 26
3.2.2 POM::BusinessClass 28
3.2.3 POM::Session 28
3.2.4 Relationships 29
3.3 Patterns of Use 29
3.4 Data Integrity Issues 31
3.4.1 Domain Constraints 32
3.4.2 Referential Integrity 32
3.4.3 Functional Dependencies 33

4 **Transaction Model in Prothos** **34**
4.1 Data Object Message Simplification 34
4.2 The Storage Hierarchy 35
4.3 The Transaction Model 36
4.3.1 The State Diagram 36
4.3.2 Model Checking The Transaction Model 39

5 **Concurrency Control** **42**
5.1 Schedules 42
5.2 Lock-Based Protocols 46
5.2.1 Locks 47
5.2.2 The Two-Phase Locking Protocol 51
5.2.3 Prothos' Optimistic Concurrency Control Mechanism . 52
5.2.4 Other Protocols 58
5.3 Model Checking Lock-Based Protocols 59
5.3.1 Model Construction 60
5.3.2 Specifications 62
5.3.3 Starvation 64
5.3.4 Hermes 66

6 **Conclusions and Future Work** **68**
6.1 Conclusions 68
6.2 Future Work 70

Bibliography **72**

A The SMV Model **76**

List of Tables

3.1 Mapping from POM Data Model to Relational Data Model . . 26

4.1 The Transaction Model's Transition Table 38

5.1 Transaction Tr_1 43
5.2 Transaction Tr_2 43
5.3 A Serial Schedule (in which Tr_1 is followed by Tr_2) 44
5.4 A Serial Schedule (in which Tr_2 is followed by Tr_1) 44
5.5 A Serializable Schedule 45
5.6 A Nonserializable Schedule 46
5.7 The Relationship Between Locks 47
5.8 A General Locking Schedule (which causes inconsistency) . . . 48
5.9 A General Locking Schedule (which causes deadlock) 49
5.10 A Sample Deadlock Situation 50
5.11 The Relationship Between Locks in POM 54
5.12 Transaction Tr_1 in Prothos 56
5.13 Transaction Tr_2 in Prothos 56
5.14 A Schedule under Prothos' Concurrency Control Scheme . . . 57
5.15 Part of the Counter Example 65

List of Figures

1.1 Study of POM in Three Layers 3

2.1 Levels of Abstraction in a Database 5
2.2 Application Developed from a Framework (adopted from [15]) 8
2.3 Prothos' Architecture 11
2.4 POM's Structure 13
2.5 Data Flow through DBI 14

3.1 A Sample Class Diagram 22
3.2 A Sample Instance Diagram 23
3.3 Class Diagram of Relational Data Model 24
3.4 Class Diagram of POM Data Model 25

4.1 The Storage Hierarchy 35
4.2 The Transaction State Diagram 37
4.3 The CTL Diagram of The Transaction Model 40

5.1 Concurrency Control in Prothos 53
5.2 State Transition of Concurrency Control in Prothos (current implementation) 60
5.3 State Transition of Concurrency Control in Prothos (future improvement) 66

Chapter 1

Introduction

1.1 Motivation

As the size and complexity of software systems increase, the overall system structure, which is also known as *software architecture*, becomes increasingly critical in software design and development.

Formal methods have a sound mathematical basis, typically given by a formal specification language. They are particularly useful for systematically describing and reasoning about computer-based systems.

Thus, an interesting question in day-to-day software engineering is that: ***whether and how formalism can help software architecture***.

To find out solutions to such a problem, we look at both structural and behavioral aspects of software architecture. Since architecture and design are broad areas, we have focused on domain-specific application frameworks. These kinds of frameworks are key to modern application development. Frameworks promote reuse and rapid development by constraining the space of possible solutions. Formal methods in constrained situations have a greater chance of being useful. Also, investment in reusable frameworks can justify investment in formal methods.

Our work centers around a software framework called "Prothos" used to create database-centric web applications. The framework is fairly mature since it has been developed, used, and evolved for a couple of years, and a number of applications have been built on top of it.

The whole software framework is a complex system. The transaction management component, named *persistent object manager* (POM), plays an important role in Prothos. It not only allows the user to define, create, and manipulate persistent business objects, but also provides transaction services to user's applications. Without POM, each application developer would have to build their own mechanism for object persistence (leading to interoperability problems between applications) and would also have to make decisions about whether to allow concurrent access and how to implement this.

It is indispensable and significant to capture the behavior of POM in Prothos. Our approach is to analyze, model, and reason about the core features in a formal way. The ultimate goal is to verify the correctness of the original design, clarify the crucial but implicit assumptions in the framework, and provide potential improvements for the current version of Prothos. Our practice is to investigate the following questions, given a mature software framework:

- Can formal methods really help us understand software architecture?
- And when should we make use of formal methods in the software lifecycle?

Our case study is to check whether this approach is worth doing, and to learn something about the problems—both formal methods and software architecture. From this experience, we could then develop a more systematic approach to further similar practice. Our experience can also provide a "road map" of how to tackle an architectural study.

1.2 Thesis Structure

The next chapter surveys related work of this thesis. We provide the background overview of our work: database management systems, software framework, and formal methods. We also review some related research and practice in the literature.

Chapters 3 to 5, as a group, present a formal way to understand the transaction management component's behavior in Prothos. The software component is formally studied through three different layers. Each chapter focuses on characteristics in one specific level. The hierarchy is shown in Figure 1.1.

Chapter 5	concurrency control
Chapter 4	single transaction model
Chapter 3	POM architecture

Figure 1.1: Study of POM in Three Layers

POM's architecture is considered in Chapter 3. We analyze several data models including the underlying POM data model. Then, we build up POM's ontology to provide the shared knowledge throughout systems and among stakeholders. Patterns of use are provided and the goal is emphasized. In particular, the issues of data integrity are addressed. The POM architecture builds up the foundations for discussion in the other two levels.

We explore the non-concurrent transaction model in Prothos in Chapter 4. Atomicity is probed in this layer. Model checking is applied in this context. This level lays the underpinning for the concurrency control of transactions in the framework.

Chapter 5 concentrates on concurrency issues. Several lock-based concurrency control protocols are introduced and formal models are developed in order to check the most considerable properties in this layer — serializability and deadlock freedom. This is a key chapter as it presents Prothos' optimistic concurrency control mechanism as well as detects the starvation problem in the framework.

Finally, we conclude in Chapter 6 with a summary of contributions and a speculation of future work.

Chapter 2

Background Practice and Related Research

2.1 DBMS

In any type of organization, considerable resources and activities are dedicated to the gathering, filing, processing, and exchanging of data based on well-established procedures in order to achieve specific goals. For example, in a university, data management systems are set up for the purpose of providing education services, whereas in a bank, data organization is based on the provision of financial services. The development of data management technology with the implementation of database systems tries to meet this need.

A *database management system* (DBMS) consists of a collection of interrelated data and a set of programs to access that data. The collection of data, usually referred to as the *database*, contains information about one particular enterprise. The primary goal of a DBMS is to provide an environment that is both *convenient* and *efficient* to use in retrieving and storing database information [25].

The objective of database systems is to manage large bodies of information. The management involves both the definition of structures and the provision of mechanisms for data manipulation. In addition, a DBMS must consider other factors such as security, distribution, concurrency, integrity, and so on.

One of the major functions a database system should provide is an abstract view of the data. It is obvious that between the computer, dealing with bits, and the user dealing with applications, there exist some levels of abstractions. A fairly standard viewpoint regarding abstractions in a database is depicted in Figure 2.1.

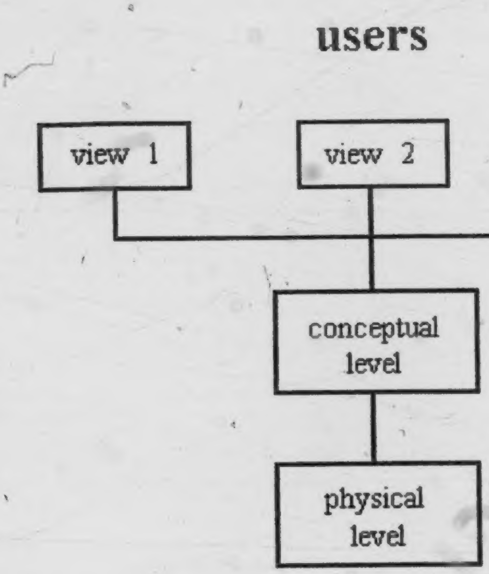

Figure 2.1: Levels of Abstraction in a Database

- The *physical* level of abstraction describes *how* the data are actually stored. The physical database resides permanently on secondary storage devices, such as disks and tapes [45].
- The *conceptual* abstraction level is responsible for defining *what* data are indeed stored in the database, and the relationships that exist among the data. Data stored in this level could be thought of as a description of enterprise information that is implementation-independent.
- A *view* is an abstract model of a portion of the entire database. This level of abstraction is necessary since users of the database system may only be concerned with part of the information.

2.1.1 RDBMS

The *relational database management system* (RDBMS) adopts the relational data model as its basis. The mathematical concept underlying the relational model is the set-theoretic *relation*, which is a subset of the Cartesian product of a list of domains.

A relational database consists of a collection of relations. The members of a relation are called *tuples*. It helps to view a relation as a *table*, where each row in a table is a tuple and each column corresponds to an *attribute*.

A relational schema is a description of the structure of relations in a relational database. The schema is a relatively static concept which defines the logical design of the database. Basically, a relation schema is a list of attributes and their corresponding domains.

RDBMS have been extensively studied in computer science, and there are many references in the literature like [1], [32], [46]. We assume that the reader has a general knowledge of RDBMS. In Chapter 3, we will discuss Prothos' data model, and further investigate the properties of RDBMS.

2.1.2 OODBMS

The *object-oriented database management system* (OODBMS) integrates database technology with the object-oriented programming paradigm.

The object-oriented data model is based on the concept of encapsulating data and code that operates on that data into a single unit called an *object*. The interface between an object and the rest of the system is defined by a set of *messages*.

An object could be abstractly viewed as a triplet: identity, state, and behavior. An object retains its identity even if its state changes over time. Its behavioral features are exhibited by responding to messages sent through the interface.

Similarly structured objects are grouped into *classes*. The set of classes is structured into a sub- and superclasses hierarchy using the concept of *inheritance*.

There are some notable differences between the relational model and the object-oriented one. Above all, complex objects cannot be modelled using a single table in relational models, but instead require additional tables and foreign keys. A complex object has a collection[1] of instance variables. The value of an instance variable can be an object or a collection of objects. This characteristic enables the complex object to have an arbitrarily complicated internal structure. In object-oriented models, the collection is an attribute of the object.

Moreover, the relational model does not provide the notion of inheritance. Additionally, there are no mechanisms for associating operations with the definition of data objects in the relational schema, and the behavioral semantics of objects is dispersed in application programs. Finally, the concept of the identity of objects does not apply to tuples of a relational database. We cannot have two identical tuples in a relation, but can have two distinct objects with equal attributes except the identity. That is, the object identity always provides a unique key.

2.2 Software Framework

A *software framework* is a reusable design and implementation for all or part of a software system. An object-oriented framework is typically implemented as a set of abstract classes, which define the core functionality of the framework along with concrete classes for specific applications.

Frameworks exploit the commonalities in design and implementation between a family of related applications. Software engineers use frameworks to reduce the cost of building complex applications.

Users of the framework complete or extend the framework by adding custom application specific components or functions to produce an application [15]. A good software framework consists of three parts: software architecture, implementation, and hooks. Each hook provides a concise solution to a focused requirement or problem within the context of the framework [14].

---

[1] A collection includes any container such as sets, lists, classes, etc.

The concepts associated with frameworks can be identified in Figure 2.2. Applications are developed on top of frameworks by filling in missing pieces and customizing the framework in the appropriate areas.

The core of the framework normally consists of abstract classes that define the generic structure and behavior of the framework. Framework libraries are extensions to the framework core comprising concrete components that can be used with little or no modification by applications. Classes in the unused library are not required by the application. Application extensions define specific classes that do not exist in the framework.

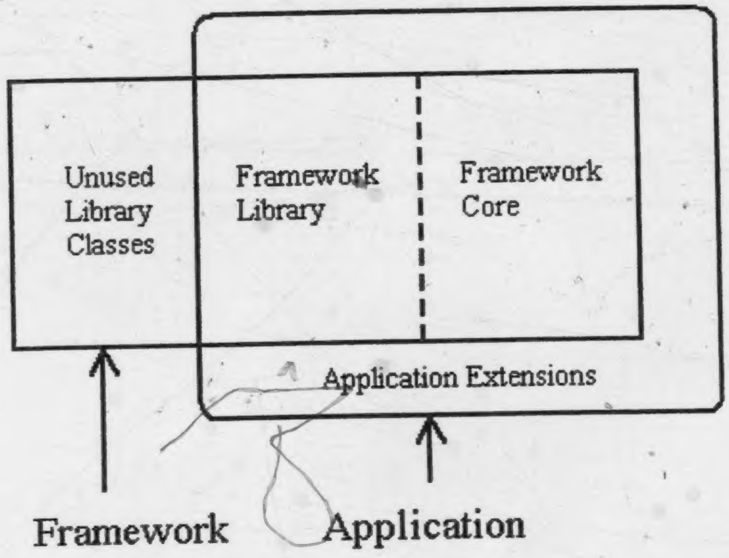

Figure 2.2: Application Developed from a Framework (adopted from [15])

Frameworks are meant to be reused to develop applications. Application-specific functional requirements are usually implemented by applications. Some non-functional properties like reusability, extensibility, and flexibility, then become dominant criteria when evaluating a framework.

2.2.1 Software Architecture

All software systems, including frameworks, must have a solid foundation at the level of *software architecture*. Software architecture studies the higher level issues involved in software design, such as the organization of a system as a composition of components, global control structures, physical distribution of subsystems, scaling and performance, selection among design alternatives, and so forth.

Two popular definitions of software architecture are proposed by Perry and Wolf [34] and Shaw and Garlan [40]. Both of them could be abstractly viewed as a triplet. In [34], software architecture is interpreted as a model that consists of: *elements*, *form*, and *rationale*. In [40], software architecture involves *components*, *connectors*, and *configurations*. No matter which definition we choose, software architecture provides us a model of the large scale structural and behavioral properties of systems.

Software architecture raises the level of abstraction at which developers can reason about their systems. A system's architecture provides a model of the system that suppresses implementation details and increases the independence of system components, permitting many issues to be localized. By suppressing these details at the architectural level, the architect can concentrate on the analyses and decisions that are most crucial to the system structure [2].

2.2.2 Prothos

Prothos is a domain-specific application framework used in creation of database-centric web applications. It is designed and developed by Avra Software Lab Inc[2].

Web applications are multi-user, with access control based on the logged-in users. However, the *Hyper Text Transfer Protocol* (HTTP) is connectionless, therefore Prothos provides full support for server-side session and page instance data. With a secure login and session management scheme, access to pages, workflow, and database records are controlled.

---

[2] More information could be found at: http://www.avrasoft.com .

Web applications support multiple concurrent transactions, and Prothos provides the optimistic concurrency control mechanism to guarantee serializability and deadlock freedom. This prevents concurrent users from taking actions based on obsolete data, such as placing an order that has been already modified by someone else during the placing time.

Since Prothos web applications are transaction based, if a failure is going to occur, it will occur safely. Prothos uses a structured exception handling mechanism. This ensures that if anything goes wrong during the processing of a transaction, no changes are made to the database and the user is informed of the failure as well as it being logged along with the relevant debug information.

Prothos applications use a three-tier architecture, in which the user interface functions, business and workflow functions, and the database management functions are delivered by relatively independent pieces of software. This separation of functionality makes it much easier to maintain web applications developed with Prothos.

The Prothos framework is written in Perl, a programming language which is available on all Microsoft and Unix-like back ends. Perl includes its own facilities for interfacing with almost any relational database management system, with full support for atomic transactions and referential integrity.

Figure 2.3 illustrates the Prothos' architecture. The acronyms and abbreviations are explained as follows:

• **UA**	User Agent (an instance of a user working with the application)
• **UI**	User Interface
• **BC**	Business Class
• **POM**	Persistent Object Manager
• **UIM**	User Interface Manager
• **SQL**	Structured Query Language

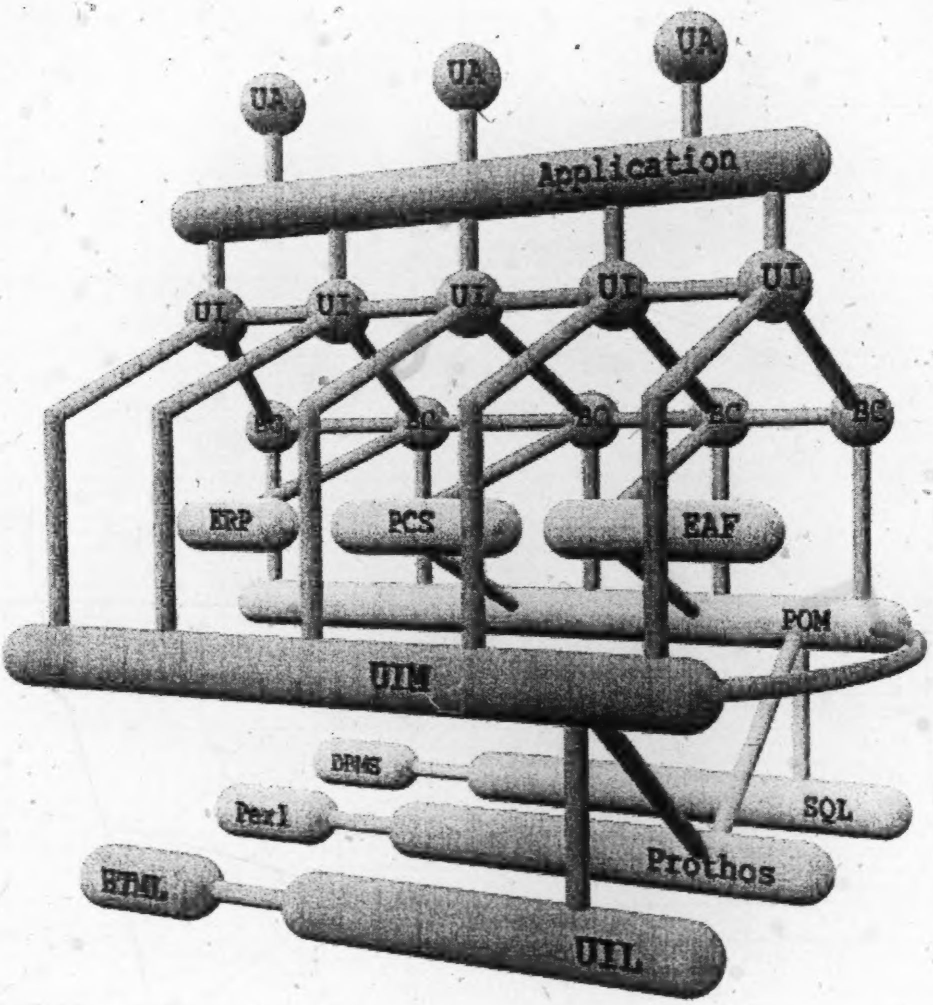
UA
UA
UA
Application
UI
UI
UI
UI
UI
BC
BC
BC
BC
BC
ERP
PCS
EAF
POM
UIM
DBMS
SQL
Perl
Prothos
HTML
UIL

Figure 2.3: Prothos' Architecture

DBMS	Database Management System
Perl	Practical Extraction and Reporting Language (*also known as:* Pathologically Eclectic Rubbish Lister)
UIL	User Interface Language
HTML	Hyper Text Markup Language
EAF*	Engineering Application Framework
PCS*	Product Catalog Service
ERP*	Enterprise Resource Planning

The components marked with asterisk signs are examples of services that one could attach to Prothos using the service hooks as required by an application. Strictly speaking, they are not part of Prothos framework.

There are many web applications built from Prothos like the *Software Engineering Research Laboratory* (SERL)[3] at the Department of Computing Science, University of Alberta, the *Alberta Software Engineering Research Consortium* (ASERC)[4], etc.

Prothos was successfully introduced to a one-term senior course to help students develop their projects [23]. The fact that all five teams[5] finished their web applications on time and in high quality demonstrates the reusability, robustness, and maturity of Prothos.

2.2.3 POM Structure

As shown in Figure 2.3, Prothos is a layered system. The whole architecture is sufficiently complicated that it is sometimes unrealistic to cover all the details when describing it to a developer. Then, what is the most important part in the architecture?

If we choose Shaw and Garlan's [40] *software architecture* definition, each

---

[3] http://peoria.cs.ualberta.ca/serl

[4] http://peoria.cs.ualberta.ca/aserc

[5] Each team consists of seven people.

"big" part in the picture is a "component", and the lines connecting different components are "connectors". The more connectors a component is associated with, the more interactions it is supposed to have, and the more important role it plays in the system.

The *Persistent Object Manager* (POM) has five different direction-oriented types of connectors, which is the largest among all the components. From this perspective, POM is the most important part in the architecture. And that is why we pay most of our attention to it.

The POM's structure is also a layered system which is shown in Figure 2.4.

POM
Perl DBI
DBD::Avra
Avra RDBMS

Figure 2.4: POM's Structure

The primary interface for database programming in Perl is *database interface* (DBI). DBI is database-independent, which means that it can work with any relational database, such as Oracle, Sybase, Informix, Access, MySQL, etc. The Perl DBI is intended for relational database, and so does not support any OODBMS [12].

DBI separates the actual *database drivers* (DBDs) from the programmer's *application programming interface* (API). Figure 2.5 illustrates the flow of data from a Perl script through to the database.

The data manipulation methods are defined within the DBI module. From there, the DBI module works out which driver should handle the execution of the method and passes the method to the appropriate driver for actual execution. In fact, the DBI module does not perform any database work itself, nor does it even know about any types of databases whatsoever.

This separation of the drivers from the DBI itself makes the DBI a powerful programming interface that can be extended to support almost any relational database available today. The Avra RDBMS is designed and developed by Avra Software Lab Inc[6] and is used in the Prothos framework.

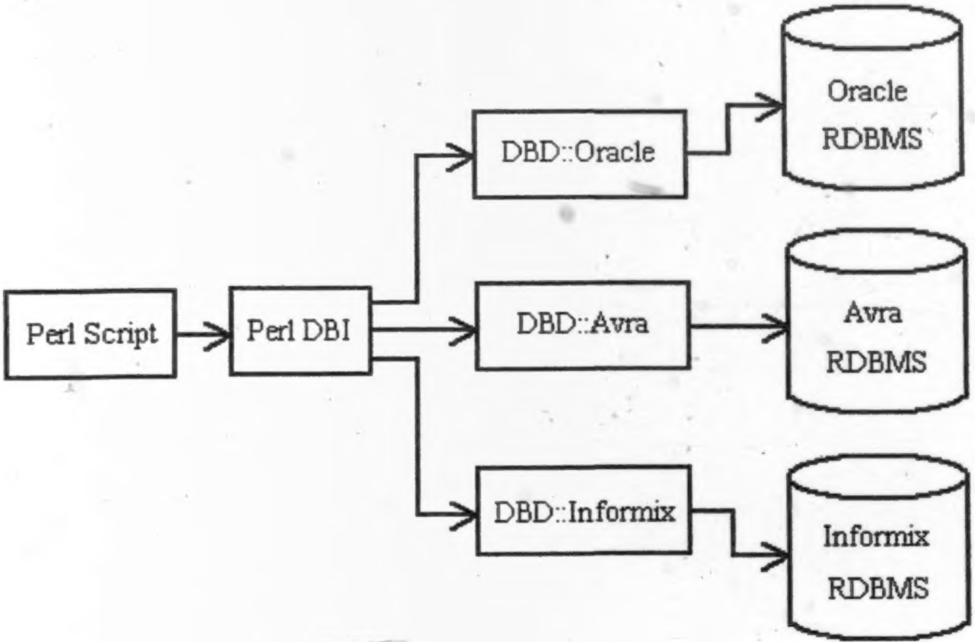
Perl Script
Perl DBI
DBD::Oracle
DBD::Avra
DBD::Informix
Oracle RDBMS
Avra RDBMS
Informix RDBMS

Figure 2.5: Data Flow through DBI

2.3 Formal Methods

Engineering relies heavily on mathematical models and calculation to make judgments about designs. This is in stark contrast to the way in which software systems are designed — with *ad hoc* techniques and ongoing testing. Formal methods bring to software and hardware design the same advantages that other engineering endeavors have exploited: mathematical analysis based on domain models [9].

In a nutshell, formal methods are the applied mathematics of computer systems engineering. They use mathematically based notions to establish models that correspond to the real systems, then to prove models are free of design errors [7]. This verifies the correctness of the abstract model, not the real system though. "Formal" means relating to or involving outward form or structure. Here, it could be further understood that the methods of reasoning are valid by virtue of their form and independent of their content.

---

[6] More information could be found at: http://www.avrasoft.com .

In [4], "formal methods" are interpreted as a generic and inadequate terminology to describe techniques (tools, languages, concepts, ...) aiming at giving more insurance on the behavior of systems before they are built and running.

Formal methods are used to specify and model the behavior of a system and to formally verify that the system design and implementation satisfy functional and safety properties. They express some behavior of the system in a "formally executable" way (the model), using either a software tool to run the model (symbolic testing) or a model checker or theorem prover to certify some properties (model checking).

Formal methods refer to the use of techniques from logic and discrete mathematics in the specification, design, and construction of computer systems (both hardware and software) and rely on a discipline that requires the explicit enumeration of all assumptions and reasoning steps. Each reasoning step must be an instance of a relatively small number of allowed rules of inference. In essence, system verification is reduced to a calculation that can be checked by a machine. In principle, these techniques can produce an error-free design [9].

Although formal methods have a sound basis on mathematics, they are often regarded as impractical [21]. Nevertheless, it is important to realize that formal methods are **not** an all-or-nothing approach. The application of formal methods to the most critical portions of a system is a pragmatic and useful strategy [9].

2.3.1 Model Checking

A *model* is a representation of a set of objects and their relationships. Such a representation is always the result of a process of abstraction.

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not.

Computation Tree Logic (CTL) is defined by Clarke and Emerson [10]. It is a subset of *temporal logic*, which is the generic term for logics that have explicit operators for making statements about changes over time [8].

CTL let us talk about how the state of a system can evolve over time. Usually in our normal operational semantics of machines, we talk about explicit sequences of states, indexed by time. CTL allows us to push the indexing details behind the scenes, but at some loss of expressive power.

CTL formulas consist of the usual atomic propositional logic formulas, plus temporal connectives. The propositional logic formulas are expressions about the state of the system. The temporal connectives are expressions about all possible paths into the future that the state of the system can follow.

Temporal connectives are pairs of symbols. They talk about what can happen from the current state. The "current" state is the one being described in the formula. The future is infinite, that is, the computation does not halt, although it can stay in the same state forever.

Connectives have the following syntax. The first member of the pair is one of:

- A – the universal quantifier, meaning on all paths from the "current" state, read as "inevitably";
- E – the existential quantifier, meaning on at least one path from the "current" state, read as "possibly".

The second member of the pair is one of:

- X – meaning the next state;
- F – meaning some future state;[7]

---

[7] In the CTL temporal logic, the current state is also esteemed a future state.

- G – meaning all future states, read as "globally";
- U – meaning until.

There are, in total, eight temporal connectives in CTL which are described below. Suppose that the system is in some state S. Consider some CTL formula ϕ: Its truth value depends on the current state S which is determined by the context in which ϕ appears.

1. $\boldsymbol{AX}\phi$ is true if and only if ϕ is true for every immediate successor to state S;
2. $\boldsymbol{AF}\phi$ is true if and only if on all paths into the future from S, there is a state where ϕ holds;
3. $\boldsymbol{AG}\phi$ is true if and only if ϕ is true for every successor to state S, including S, i.e. ϕ is true for all states on all paths into the future from S.
4. $\boldsymbol{A}(\phi \boldsymbol{U} \psi)$ is true if and only if all paths starting in state S satisfy ϕ until reaching a state in which ψ holds;
5. $\boldsymbol{EX}\phi$ is true if and only if ϕ is true for at least one immediate successor to state S;
6. $\boldsymbol{EF}\phi$ is true if and only if there exists a path into the future from S on which there is a state where ϕ holds;
7. $\boldsymbol{EG}\phi$ is true if and only if there is a path from S into the future for which ϕ holds for every state on the path, including S;
8. $\boldsymbol{E}(\phi \boldsymbol{U} \psi)$ is true if and only if there exists a path starting in state S that satisfies ϕ until reaching a state in which ψ holds.

Thus, past time operators are not allowed, and temporal operators cannot be combined directly with the propositional connectives. Note that since all operators are prefixed by A or E, the truth or falsehood of a formula depends only on the given state, and not on the particular branches in the temporal tree that were followed to get to the state. From this perspective, CTL is memoryless.

The *Symbolic Model Verifier* (SMV) system is a tool for checking finite-state systems against specifications in the temporal logic CTL. The input language of SMV is designed to allow the description of finite-state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract [30].

Some pointers are given here, [8] [10] [30], for further reference. They are all related to model checking, temporal logic, CTL, or SMV.

2.4 Related Work

Formal methods are such powerful tools that people in computer science have realized their importance since 1970s. Many projects have applied formal methods to software architecture. For example, IBM's CICS system contains more than a half million lines of code. IBM is using Z to respecify key CICS interfaces to improve its maintainability [33]. These activities are trying to address formal underpinnings for architecture. As new notations are developed, and as the practice of architectural design is better understood, formalisms for reasoning about architectural designs become relevant [40].

Efforts to formalize software architecture began with efforts to apply formal methods directly to individual systems or styles. More recently, there have been some more general treatments of architectural description and analysis. A number of *Architecture Description Languages* (ADLs) and their accompanying tool sets have been proposed as modelling notations to support architecture-based development such as Aesop [16], Darwin [29], Rapide [28], UniCon [39], and Wright [2].

Recently, the work has been done on an architecture interchange language, ACME [17], which is intended to support mapping of architectural specifications from one ADL to another and, hence, enable integration of support tools across ADLs. Although, strictly speaking, ACME is not an ADL, it contains many ADL-like features. Medvidovic and Taylor [31] presented a classification and comparison framework for ADLs to capture highlights in this area.

Before ADLs became popular in the past few decades, research in this field had been developed for a long time. *Module Interconnection Languages* (MILs) and *Interface Definition Languages* (IDLs) were centers of attraction during 1980s. MILs, such as MIL75 [11] and INTERCOL [44], are intended to support the description of large-scale program structure independent of any particular programming language or system. However, MILs are *not* concerned with what the system does (specification information), how the major parts of the system are embedded into the organization (analysis information), or how the individual modules implement their function (detailed design information) [35].

IDLs have been developed to define abstract data interfaces and mappings from programming language constructs into intermediate representations that mediate the transformation between incompatible low-level formats. Examples of these languages include IDL [27], UTS [22], and CORBA's IDL [19]. These languages define a single, restricted model of interaction, typically procedure call or object-method invocation. However, they do not provide a sufficient general basis for the description and analysis of interactions in software architectures.

Although ADLs are definitely a promising direction, it is different from our approach. Instead of defining a descriptive language for the whole software architecture, we are applying formal methods in a domain-specific application framework. The formal models of the architecture in the design phase can be used as a basis for verification of further implementations. We pay our attention to the transaction management component in a specific framework, not the entire architecture. We are intended to present a formal way to understand software architecture in this context at both structural and behavioral aspects.

Another related work is Microsoft's SLAM project [37]. The goal of the project is to check system programs written in C against safety properties using model checking techniques. SLAM introduces a new type of programs called "boolean program" which has only boolean variables in it. The boolean program can take full advantage of *binary decision diagram* (BDD) technique used in model checking. Although boolean program model has proved itself and has been successfully used for checking device drivers, one challenge is how to bridge the gap from a language with pointers (C) to one without pointers (boolean programs). Another problem is the "state space explosion" problem of model checking when it is used in the system programs' context. We do not employ any of the above techniques: boolean program, BDD, etc. We simply build "normal" SMV models based on the current implementation of the framework.

With respect to the software framework, an interesting research is *Framework Constraints Language* (FCL) [24]. Framework constraints are rules that frameworks impose on the code of framework-based applications. FCL helps formally specify these constraints, and further automatically detect errors occurring at the boundary between frameworks and applications. Given the Prothos framework, our formal approach to understanding the transaction management component's behavior would certainly generate a number of crucial framework constraints to help develop Prothos' FCL.

2.5 Thesis Scope

Our work focuses on the transaction management component in the Prothos framework. And among formal methods, we mainly concentrate on model checking technique. By providing a precise semantics for the component at the abstract level, formal models of the system can supply the basis for rigorous, justifiable analysis of critical system properties.

Chapter 3

POM Architecture

We explore the *POM architecture* in this chapter to investigate the goals, structure, and design decisions of this component in the framework. We first focus on the basic data model of POM and how this model is elicited. Then, we build up POM's ontology to provide the consistent terminology throughout the software framework and applications and among the stakeholders. Patterns of using POM are provided and the goal is emphasized. Operations in the basic database level include: query, insertion, deletion, and update. Data integrity constraints provide a means of ensuring that changes made to the database using these basic operations do not lead to a loss of data consistency. A sequence of operations is encapsulated into a transaction.

3.1 Data Models

3.1.1 Object-Oriented Data Model

The object-oriented paradigm is one approach to modelling the data and operations of a computation. The object-oriented approach is ideally suited for many classes of real world problems, especially those having workflow aspects.

There are three fundamental concepts defined in object-oriented paradigm: *object*, *class*, and *inheritance*. An *object* is an entity with a public protocol and a private structure. A *class* is an abstraction of common protocol and a mechanism for object creation. An object is an instance of a class. *Inheritance* captures the commonality hierarchy between classes, and provides shared protocols between objects from different classes in the hierarchy.

By using the object-oriented model, a program consists of a set of messages sent to objects. A problem is solved by sending a message to an object and getting the solution object as a response. Objects have state so they can remember information. Messages are implemented by methods that access state or send messages to other objects [43].

To illustrate different data models, consider a concrete example using *unified modeling language*(UML) [38]. Figure 3.1 depicts the class diagram expressing an association relationship between courses and sections. Each section object is associated with a course. A course can be associated with multiple section objects.

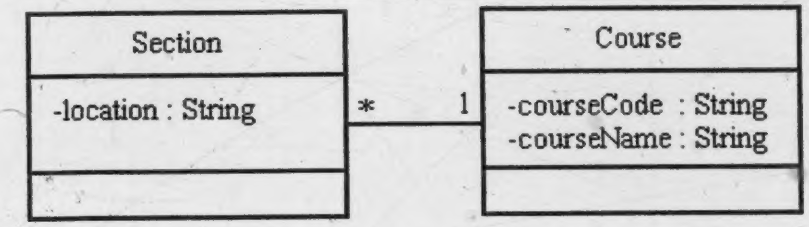

Figure 3.1: A Sample Class Diagram

The class diagram could also be called the object schema. Class diagrams show all the possible relationships. An instance diagram illustrates a specific example of objects satisfying the class relationships.

Figure 3.2 is an instance diagram based on Figure 3.1. An instance diagram uses a real world example, checks the validity of the class diagram, and exposes defects in the model. It shows that the "Algorithms" course (course code is "CMPUT 204") corresponds to three sections whose locations are in "CSC B10", "CAB 377", and "GSB 859" respectively.

3.1.2 Relational Data Model

The mathematical concept underlying the relational model is the set-theoretic *relation*, which is a subset of the Cartesian product of a list of domains. A

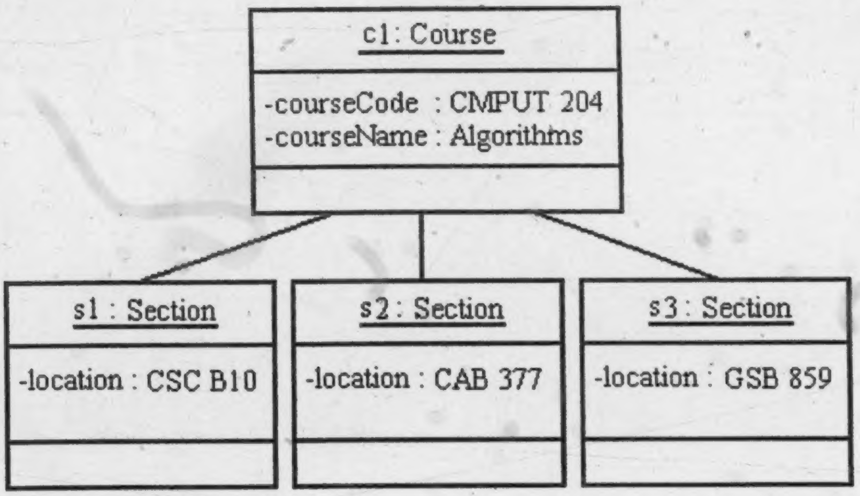

Figure 3.2: A Sample Instance Diagram

relational database consists of a collection of relations. The members of a relation are called *tuples.* It helps to view a relation as a *table*, where each row in a table is a tuple and each column corresponds to an *attribute*.

A relational schema is a description of the structure of relations in a relational database. The schema is a relatively static concept which defines the logical design of the database. Basically, a relation schema is a list of attributes and their corresponding domains. The data values of the relations in the database may change from time to time.

Figure 3.3 shows the UML class diagram of a relational database. There is a "one-to-one correspondence" relationship between the "Table schema" class and "Table" class. A table is created or destroyed by sending messages to a table schema object. Each table contains one or more columns. Rows are manipulated within a table. All four basic database operations, namely query, insertion, deletion, and update, are all defined in the "table" class. A relational data model is nothing but flat tables.

Object-oriented is a programming notion, and an object-oriented model is normally transient and exists only during execution. To allow the model to persist, an *object-oriented database management system* (OODBMS) is re-

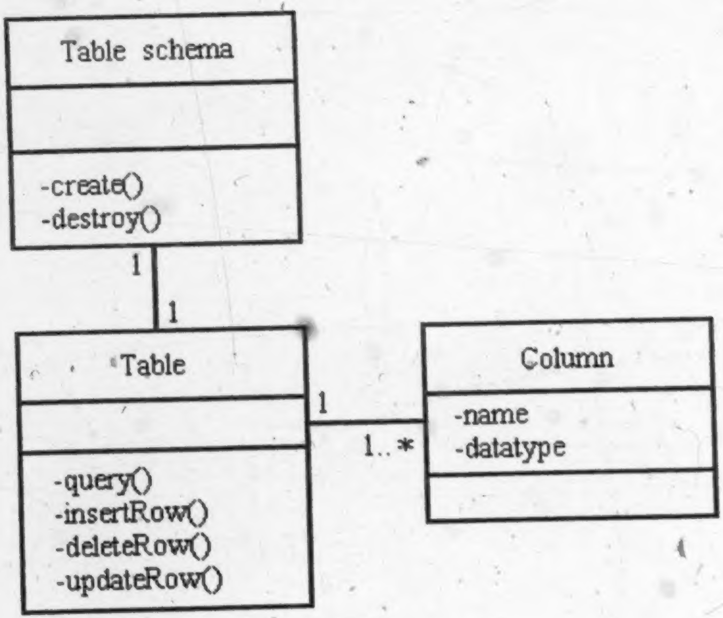
Table schema
-create()
-destroy()
1
1
Table
-query()
-insertRow()
-deleteRow()
-updateRow()
1
1..*
Column
-name
-datatype

Figure 3.3: Class Diagram of Relational Data Model

quired.

An OODBMS integrates database technology with the object-oriented paradigm which was discussed in the prvious section. An OODBMS must consider inheritance and versions of methods. However, there is no real production quality implementation of a pure OODBMS yet.

3.1.3 POM Data Model

POM data model is a *network* model, which can be regarded as an object-oriented model without inheritance and method storage. The associations are transparent in the network model.

A network model consists of a collection of data objects which are connected to one another through links or pointers. Each object is a collection of attributes, each of which contains a data value or an object. A link is an association between precisely two data objects.

The advantage of a network model is that the classes can be easily mapped to relations, and the objects mapped to tuples in the relation.

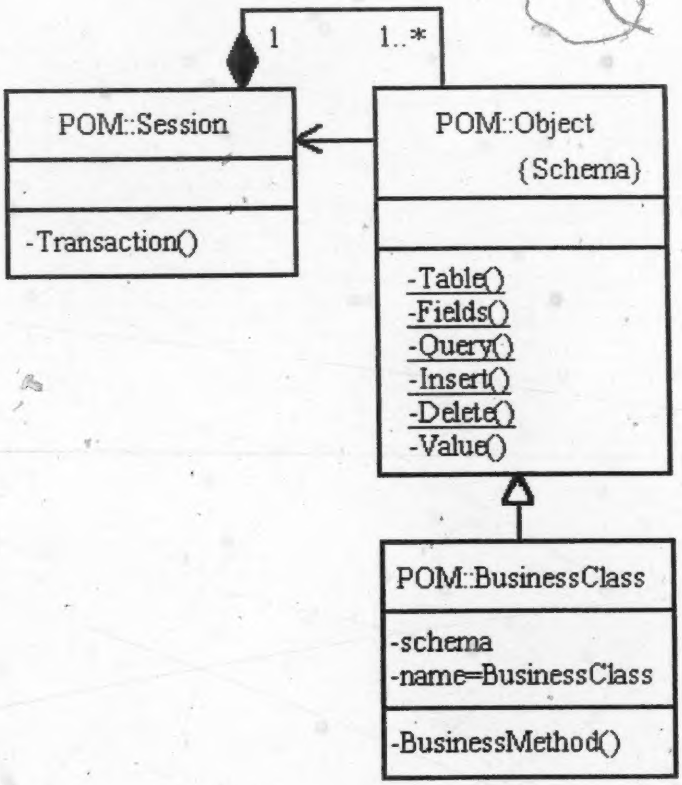
1
1..*
POM::Session
-Transaction()
POM::Object
{Schema}
-Table()
-Fields()
-Query()
-Insert()
-Delete()
-Value()
POM::BusinessClass
-schema
-name=BusinessClass
-BusinessMethod()

Figure 3.4: Class Diagram of POM Data Model

Figure 3.4 shows the class diagram of POM data model. Basically, the POM data model wants to take full advantage of the object-oriented paradigm and database storage mechanism. Thus, OODBMS is an ideal option for POM. However, since there is no real implementation of OODBMS yet, the underlying storage mechanism of POM is adopted from the relational model. So, the goal of POM is to manipulate data objects as in object-oriented programs but they are actually kept around as in relational databases.

3.2 Ontology

In artificial intelligence, an ontology is "the specification of conceptualizations" [20]. It deals with the entities and concepts in the system of interest, as well as the relationships that hold among them. The POM's ontology is the foundation of the approach to understanding the component. It provides the consistent terminology throughout the software framework and applications and among the stakeholders.

Prothos is a database-centric web application framework. It helps developers build their web applications. An application is a collection of business objects and a mechanism for issuing messages to the business objects to change their states.

Table 3.1: Mapping from POM Data Model to Relational Data Model

POM Data Model	Relational Data Model
class	relation
object	row

Table 3.1 provides us the mapping relationship from the POM data model to the conventional relational one. This mapping helps users better understand the terminology used in POM. The "POM::Object" and "POM::BusinessClass" classes are used to define and create business classes and objects. And the "POM::Session" class is used to manipulate the business objects.

The basic idea of POM's object manipulation is to talk to a class to create, locate, and destroy objects. And the modification of the state(s) of an object is defined by talking to the object. All operations must be done inside a transaction in order to guarantee the data consistency.

3.2.1 POM::Object

Conceptual description: POM::Object is the base class of all business classes in POM. It is used to define and create persistent objects that know how to save themselves to persistent storage. This class uses the POM::Session class to interface with the database.

Attribute: The class-scope "Schema". This attribute defines a certain inherent fields of all POM objects, such as: identity(ID), the optimistic locking concurrency control tag (_Delta), creation time (_Created), last update time (_Modified), and so on. The names in parentheses are fields' names appeared in POM::Object.

Methods:

- Table(): A class method to get or set the name of database table for a persistent class. The specified parameter is used to name the database table.
- Fields(): A class method to get or set the field definitions for a persistent class. When defining a database table, the parameter of this method acts as a table schema in the relational database management system.
- Query(): A class method to return a set of persistent objects in an array satisfying the selection condition(s). The basic format of query conditions in POM is: *where P sort by S*. The *where* clause is comprised of a predicate involving fields of the persistent class. And the *sort by* clause causes the objects in the result array of this query to appear in sorted order.
- Insert(): A class method to create a new object for the given class to be inserted into the database.
- Delete(): A class method to delete an existing object from the given class in the database. Note that this method checks referential integrity before deleting the object, that is, an exception will occur if the object being deleted is referenced by other object(s). We will be further investigating data integrity issues in Section 3.4.
- Value(): An instance method to return or update the specified field of an object. Compared to the relational data model shown in Figure 3.3, the update method in POM is an instance method, which means a particular

data object has to be identified before its field(s) can possibly be changed. In the relational model, the update method is sent to table directly, which is a class method.

Notice that all basic database operations are defined within a POM::Session transaction method. In other words, Query(), Insert(), Delete(), and Value() methods must be called within a transaction in order to guard the data integrity and system constraints.

3.2.2 POM::BusinessClass

Conceptual description: POM::BusinessClass inherits from POM::Object class to define and create persistent business classes.

Attributes:

- schema: The business class description. The schema defines fields' names and types of the business class. Since the class-scope "Schema" attribute defines inherent fields for all POM objects, the "schema" attribute here is an extension of "Schema". Notice that in both "Schema" and "schema" attributes, each field must have the *name* and *type* information.

- name: The name of the business class, or from the relational model's standpoint, the table name or relation name.

Method: BusinessMethod(). This allows users to define business method(s) corresponding to this particular business class.

3.2.3 POM::Session

Conceptual description: POM::Session class provides the manipulation mechanism for POM objects. A transaction is a sequence of basic database operations that performs a single logical function. Operations defined within a transaction are inseparable. A transaction must guarantee the *atomicity* requirement, that is, all operations inside a transaction happen successfully or nothing happens.

Method: Transaction(). The method is used to define and execute a transaction in order to manipulate POM objects. The sequence of basic database operations of the transaction are defined in the method body.

3.2.4 Relationships

In Figure 3.4, there are three relationships among classes: inheritance(the triangle arrow), composition(the dark diamond arrow), and navigation(the arrow).

The inheritance relationship means that every business class is a POM object. Composition indicates that each POM::Session object has one or many POM objects. These objects are involved in this session's current transaction. Navigation represents that a POM object knows which POM session it belongs to.

3.3 Patterns of Use

A software framework imposes many patterns when using it to development applications. Given the discussion of POM ontology in Section 3.2, we generalize some patterns of using POM and provide a code-based example in order for users to better understand these patterns.

1. Users must define a POM class before using it. When defining a POM class, users must provide the name and type information for each field of the class. This is embodied in the "schema" attribute.

2. The "name" attribute of business classes is unique. Different business classes must have different names.

3. In the "schema" attribute of business classes, name of fields is unique. Besides, no field can have the same name as the ones defined in "Schema" attribute of POM::Object class.

4. All basic database operations, namely query, insertion, deletion, and update, must be called within a transaction procedure.

5. The query, insertion, and deletion messages are sent to classes while the update message is sent to objects.

Based on the course-section example introduced in Section 3.1, we give a detailed code-based example to help understand the patterns of use. The following code fragment defines the *Class::Course*[1] class:

```
                    ⋮

# Class definition
package Class::Course;
use POM::Object;
@Class::Course::ISA = “POM::Object”;
Class::Course → Table(“course”);
Class::Course → Fields([CourseCode ⇒ “S”[2]],
                       [CourseName ⇒ “S”]);

                    ⋮
```

The following code fragment defines the *Class::Section* class, given the *Class::Course* class has already been defined in the system:

```
                    ⋮

# Class definition
# The Class::Course has already been defined
package Class::Section;
use POM::Object;
@Class::Section::ISA = “POM::Object”;
Class::Section → Table(“section”);
Class::Section → Fields([Course ⇒
                              “Class::Course”],
                        [Location ⇒ “S”]);
```

---

[1] “Class::” is the conventional prefix for all business classes in the Prothos framework [42].
[2] “S” is the abbreviation of type “String” defined in the Prothos framework [42].

⋮

Suppose we have inserted several data objects to *Class::Course* and *Class::Section* classes already. Note that all insertion operations must be called within a transaction. Due to renovations to the GSB building, all sections scheduled in "GSB 859" are now moved to "CSC B02". The following code fragment illustrates how to perform such a task inside a transaction:

⋮

```
# Start of a transaction
# Select all sections located in “GSB 859”
my @sections = Class::Section → Query(
                    [Where ⇒ “Location=‘GSB 859’ ”],
                    [SortBy ⇒ “Course”]);
# Update all selected sections with new location
foreach my $section (@sections) {
    $section → Value(“location”, “CSC B02”);
}
# End of a transaction
```

⋮

To conclude this section, we emphasize the goal of POM again. POM is to manipulate data objects as in object-oriented programs but they are actually kept around as in relational databases.

3.4 Data Integrity Issues

Integrity constraints ensure that changes made to the database do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates are too costly to test. Thus, we usually limit ourselves to integrity constraints that can be tested with minimal overhead.

3.4.1 Domain Constraints

We have argued earlier in Section 3.2 that users must provide the name and type information for each field of the class. In other words, a domain of possible values must be associated with each attribute. When a POM class is defined in Prothos, the type information is tightly connected with the field name.

Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data object is entered into the database, or the user wants to change the value of some existing data objects.

In Prothos' context, domain constraints would further assist us with restricting the use of data objects correctly. For instance, it would be meaningless to send an "arithmetic multiplication" message to a "String" typed data field.

3.4.2 Referential Integrity

In the POM model, referential integrity means that all pointers or links resolve. In another word, an data object cannot be deleted if some other objects point to it.

POM has offered its own way to declare the referential integrity as explained in Section 3.2 and Section 3.3. Database modifications can cause violations of referential integrity. POM guards the constraint by disallowing the data object to be deleted if there exists any data object that has a link to it. In another word, as long as associations are maintained by links, referential integrity is assured.

3.4.3 Functional Dependencies

Functional dependencies are a generalization of key dependencies. They require that the value for a certain set of fields determines uniquely the value for another set of fields.

Let $\alpha \subseteq schema\ of\ a\ business\ class$ and $\beta \subseteq schema\ of\ a\ business\ class$. The *functional dependency*:

$$\alpha \rightarrow \beta$$

holds on the business class if for all pairs of objects o_1 and o_2 in the business class such that $(o_1 \rightarrow Value(\alpha) = o_2 \rightarrow Value(\alpha))$, it is also the case that $(o_1 \rightarrow Value(\beta) = o_2 \rightarrow Value(\beta))$.

In order to understand the concept of functional dependencies, consider the schema:

$$Club\text{-}schema = (club\text{-}name,\ city,\ province).$$

Since each city is associated with precisely one province[3], the following functional dependency holds:

$$city \rightarrow province$$

Whenever the city of "Edmonton" appears as the *city* field's value in an object, the value of the *province* field must be "Alberta" of the same object.

Thus, functional dependencies allow us to express facts about the enterprise that we are modelling with the database. In Prothos, it is the responsibility of **business class methods** to protect this kind of integrity constraints.

---

[3] In this example, no two cities have the same name.

Chapter 4

Transaction Model in Prothos

A *transaction* is a sequence of operations that performs a single logical function in a database application. The basic database operations are query, insertion, deletion, and update introduced in Chapter 3. For a single transaction, the most prominent property is **atomicity**.

The *atomicity* is also known as the "all or none" requirement. The basic operations are gathered together to form a transaction because they carry out one single task and are inseparable. An example is funds transfer, in which one account is debited and another is credited. Clearly, it is vital that either both the credit and debit occur, or that neither occurs. That is, the funds transfer must happen in its entirety or not at all.

This chapter deals with non-concurrency issues. The concurrent transactions will be discussed later in Chapter 5.

4.1 Data Object Message Simplification

The basic database operations presented in Chapter 3 include query, insertion, deletion, and update. However, since transactions are programs, it is computationally difficult to determine the exact operations that a transaction performs and how operations of various transactions interact.

For this reason, the basic operations could be simplified to **read** and **write**. This simplification makes sense because ultimately each transaction can be viewed as a read/write sequence.

In the Prothos transaction model, we adopt the notations "$\mathbf{read}(O_i)$" and "**write**(O_i)" to represent the "read" and "write" messages sent to an arbitrary data object O_i respectively.

4.2 The Storage Hierarchy

The Prothos storage hierarchy is shown in Figure 4.1.

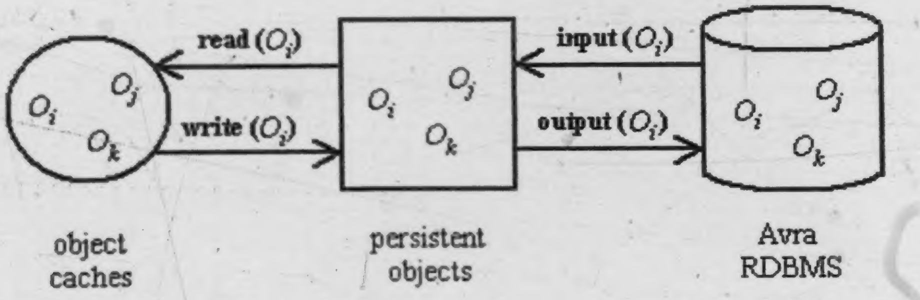

Figure 4.1: The Storage Hierarchy

The objects in object caches are actual objects in memory manipulated by the object-oriented programs. They form the application, which is a collection of business objects and a mechanism for issuing messages to the business objects to change their states.

The program talks to POM to obtain data objects, and further send them back. POM is depicted in Figure 4.1 labeled as persistent objects. The Avra RDBMS is the bottom layer used to store the data objects in the database.

Two pairs of operations transfer data objects between different storage layers.

- **input**(O_i), which transfers the physical block in which data object O_i resides to the persistent objects storage.
- **output**(O_i), which writes O_i back to the RDBMS layer.
- **read**(O_i), which makes a copy of the data object O_i to the local work space in order for the transaction program to handle.

- **write**(O_i), which assigns the local copy of O_i back to the persistent objects layer.

Notice that Prothos does a mapping between the Avra RDBMS and the object-oriented persistent layer as indicated in Section 2.2.3.

4.3 The Transaction Model

A transaction is a program unit that accesses and possibly updates various data objects. Each transaction is a unit of atomicity. In Prothos, transactions must not violate any database consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates.

4.3.1 The State Diagram

The finite-state diagram corresponding to a transaction is shown in Figure 4.2.

It is fairly straightforward to recognize that the transaction model illustrated in Figure 4.2 is a *nondeterministic finite automaton* (NFA). The formal definition chosen from [18] is:

$$M = (Q,\ \Sigma,\ \Delta,\ q_0,\ F)$$

where M stands for "machine", i.e., the transaction model in our case. The five-tuple components are specified as follows:

1. Q— a finite, nonempty set of states. In this model, the states

   $$Q = \{active,\ partially\ committed,\ failed,\ committed,\ aborted\}$$

   The following descriptions render the semantics of each state:

   - **Active**, the initial state. A transaction is in the active state when it has started execution.
   - **Partially committed**, after the last statement has been executed.

- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
- **Committed**, after "successful" completion.

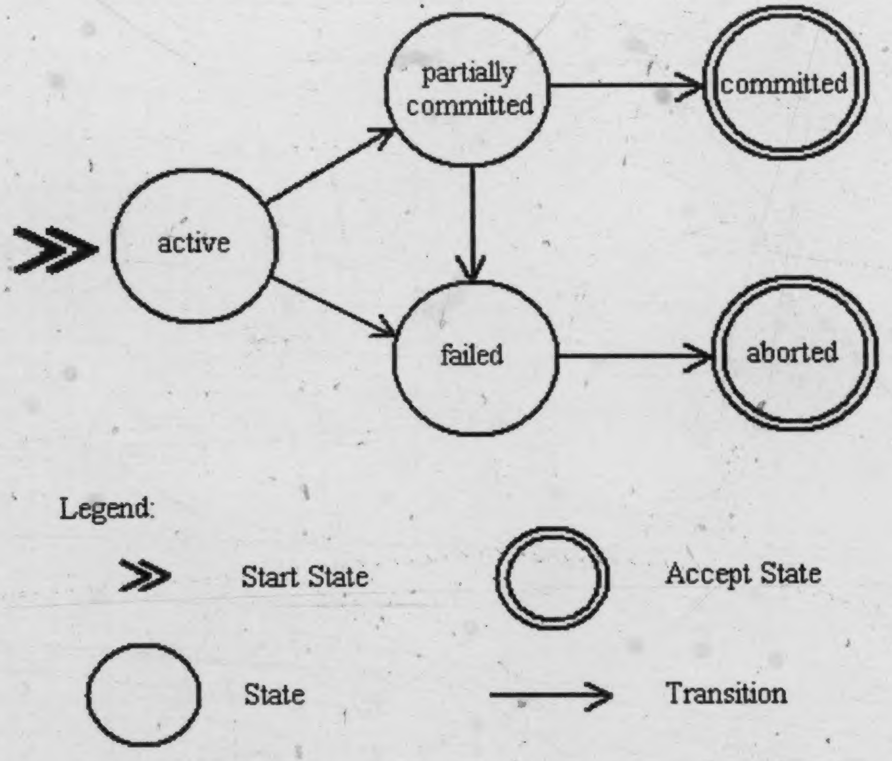
partially committed
committed
active
failed
aborted
Legend:
Start State
Accept State
State
Transition

Figure 4.2: The Transaction State Diagram

2. Σ— the data alphabet. There is no input character defined in our machine explicitly. Transactions occur between states as a result of time passing and events occurring during execution.

3. Δ— the transition relation, which is a finite subset of

$$Q \times \Sigma^* \times Q$$

Table 4.1 designates Δ for the above NFA. A transition table is a convenient method for representing the transition relation of an NFA. The transitions are numbered for convenience, but this numbering is not part of the transition relation.

Table 4.1: The Transaction Model's Transition Table

Transition Number	State	New State
1	active	partially committed
2	active	failed
3	partially committed	failed
4	partially committed	committed
5	failed	aborted

Note that we ignore the **"input symbol"** in Table 4.1 since the data alphabet, Σ, is implicitly defined over the clock tick. The time unit of any transition is greater or equal to one, indicating arbitrary input (any non-zero time interval) transitions are incorporated in this model.

Although Λ-transitions are allowed in NFAs, they are prohibited by our machine. A transaction is in and only in one state at a time, and cannot enter to another state without consuming a certain period of time.

4. q_0— the initial state or start state, $q_0 \in Q$. As we can see in Figure 4.2, the initial state is *active*, i.e., $q_0 = active$.

5. F— the set of accepting states, $F \subseteq Q$. $F = \{committed, aborted\}$ in our model. These two states do not have any "next state". This implies that once a transaction enters into either state, it terminates immediately, whether the transaction is successfully committed, or totally rolled back. This actually assures the "atomicity" property of a transaction.

Based on this model, every transaction starts in the *"active"* state. A transaction moves into the *"partially committed"* state when it reaches its last statement. At this point, all the operations have been done for this transaction, but it is still possible that the transaction may have to be aborted since the actual output may not have been written to the persistent storage yet.

A transaction goes into the *"failed"* state after it is determined that the normal execution cannot proceed any longer. Such a transaction must be rolled back, and then finishes up with the *"aborted"* state. In Prothos, an error exception is generated at this point by the framework, and it is completely the choice of the user to restart the transaction or simply kill it.

Finally, if the transaction ends up with either *"committed"* or *"aborted"* state, it terminates properly. Such a *trace* is accepted by our NFA machine, and the transaction itself does not interfere with any database constraints in that among the whole state space, only these two states preserve the system consistency. When the machine stops in a nonaccepting state, it signifies that such a transaction is rejected. In another word, the Prothos' transaction model does not permit this kind of transactions to happen as they may destroy the system consistency constraints.

4.3.2 Model Checking The Transaction Model

Model checking is an appropriate method to reason the behavior of the transaction model. We employ the *computation tree logic* (CTL), and build up a *symbolic model verifier* (SMV) model in order to observe crucial specifications of Prothos' non-concurrent transaction model. Figure 4.3 illustrates the fundamental ideas of model construction.

The important and interesting specifications of the SMV model are listed as follows:

$$AG(state = failed \rightarrow AX(state = aborted)) \tag{4.1}$$

Spec 4.1 reveals that for all nodes in the tree, the next state of "failed" must be "aborted".

$$AG(AF(state = committed \mid state = aborted)) \tag{4.2}$$

Spec 4.2 states that every branch has either "committed" or "aborted" as its ending state.

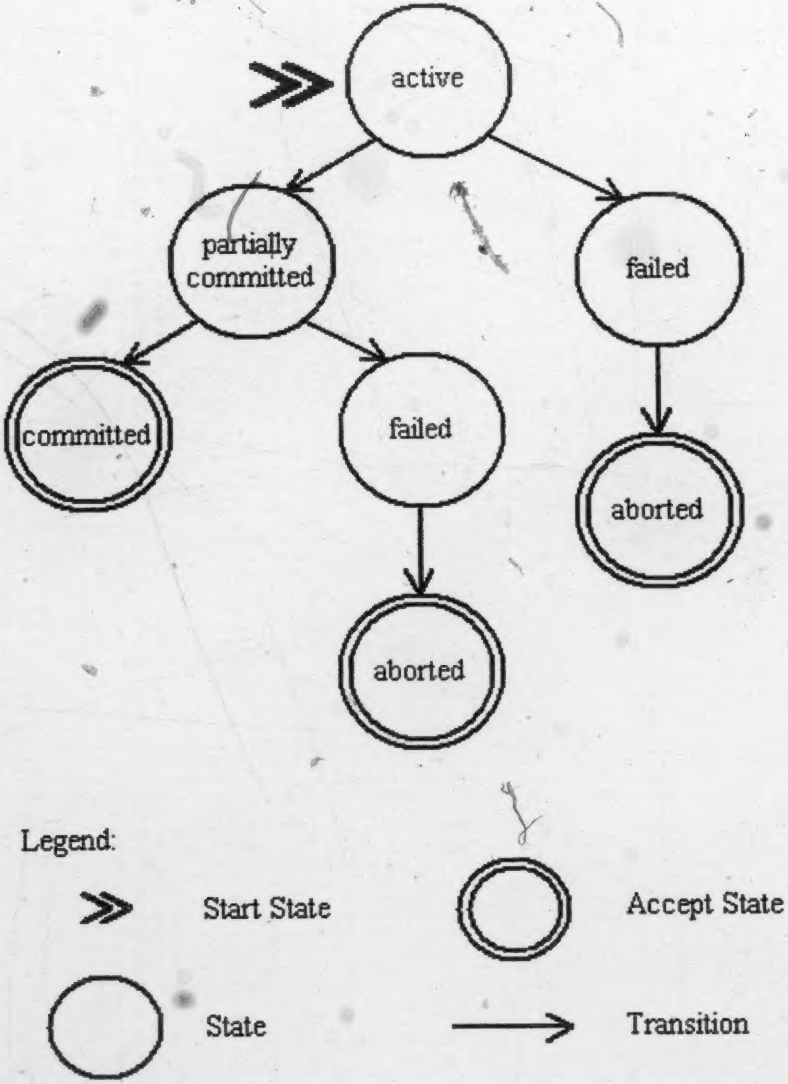
active
partially committed
failed
committed
failed
aborted
aborted
Legend:
Start State
Accept State
State
Transition

Figure 4.3: The CTL Diagram of The Transaction Model

$$AG(state = failed \rightarrow AG!(state = committed)) \tag{4.3}$$

$$AG(state = committed \rightarrow AG!(state = failed)) \tag{4.4}$$

The conjunction of **Spec 4.3** and **Spec 4.4** proclaims that "failed" and "committed" states cannot appear in one computational branch.

$$AG(state = aborted \rightarrow AG!(state = committed)) \tag{4.5}$$

$$AG(state = committed \rightarrow AG!(state = aborted)) \tag{4.6}$$

And the conjunction of **Spec 4.5** and **Spec 4.6** indicates that "aborted" and "committed" states contradict each other in any branch.

Although the above specifications provide us a formal description of the properties in the Prothos' transaction model, it is easier to extract same possessions from the model straightly using natural languages. The model has only seven states and six transitions, which is too simple to bother the model checking technique.

In consequence, model checking does not fully come into play in the context of the non-concurrent single transaction model. In other words, the basic transaction model is designed to be obviously correct to developers.

Chapter 5

Concurrency Control

It is undoubtedly necessary for Prothos to provide the concurrency control among transactions that are executed simultaneously since a web application on top of Prothos could ordinarily have many clients access the service at the same time.

The benefits of a concurrent-transaction-supportive environment are increased processor utilization and higher total transaction *throughput* — that is, the amount of work which is accomplished in a given period of time.

We have already considered Prothos' basic transaction model's atomicity in the absence of concurrency in Chapter 4. Nevertheless, when several transactions execute concurrently, the consistency of the database may no longer be preserved. In this chapter, we pay our attention to how Prothos controls the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the system.

5.1 Schedules

When a number of transactions run concurrently, database consistency can be violated despite the correctness of each individual transaction. In this section, we review the concept of *serializability* to help identify those executions that guard the system consistency.

Consider a simple banking scenario: there are two accounts A and B with the original balances of \$100 and \$200 respectively. Let Tr_1 and Tr_2 be two transactions that transfer funds from one account to another. Transaction Tr_1

transfers \$20 from account A to account B and is defined in Table 5.1.

Table 5.1: Transaction Tr_1

Tr_1 :	read(A)
	A = A - 20
	write(A)
	read(B)
	B = B + 20
	write(B)

Transaction Tr_2 transfers \$40 from account B to account A and is defined in Table 5.2.

Table 5.2: Transaction Tr_2

Tr_2 :	read(B)
	B = B - 40
	write(B)
	read(A)
	A = A + 40
	write(A)

A *schedule* is an execution sequence that stands for the chronological order in which basic operations are performed in the system. Suppose that these two transactions are executed concurrently in the order Tr_1 followed by Tr_2. Table 5.3 exhibits such a schedule with chronological order from top to bottom. We use the convention that operations of Tr_1 appear in the left column and those of Tr_2 are to the right.

After the execution in Table 5.3, the eventual values of accounts A and B become \$120 and \$180 respectively. In this example, the total amount of money in A and B should remain unchanged after the execution of both transactions. That is to say the preserved sum, $A + B =$ \$300, is the consistent constraint in this case.

A schedule for a set of transactions must consist of all operations of those transactions, and must maintain the order in which the operations appear in

Table 5.3: A Serial Schedule (in which Tr_1 is followed by Tr_2)

Tr_1	Tr_2
read(A)	
A = A - 20	
write(A)	
read(B)	
B = B + 20	
write(B)	
	read(B)
	B = B - 40
	write(B)
	read(A)
	A = A + 40
	write(A)

each individual transaction.

The schedule displayed in Table 5.3 is a *serial schedule*. Each *serial schedule* consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule [25]. Thus, for a set of n transactions, there are $n!$ valid serial schedules in total.

Table 5.4: A Serial Schedule (in which Tr_2 is followed by Tr_1)

Tr_1	Tr_2
	read(B)
	B = B - 40
	write(B)
	read(A)
	A = A + 40
	write(A)
read(A)	
A = A - 20	
write(A)	
read(B)	
B = B + 20	
write(B)	

Table 5.3 delineates one of the two serial schedules in our sample. The other one is shown in Table 5.4. Again, the sum $A + B$ is preserved as expected.

When several transactions are executed concurrently, operations from various transactions could be interleaved. In this case, the schedule is not serial any more. Thus, the number of possible schedules for a set of n transactions is much larger than $n!$. The schedule in Table 5.5 is not a serial schedule. However, after the execution takes place, the sum $A + B$ is indeed preserved.

Table 5.5: A Serializable Schedule

Tr_1	Tr_2
	read(B)
	B = B - 40
	write(B)
read(A)	
A = A - 20	
write(A)	
	read(A)
	A = A + 40
	write(A)
read(B)	
B = B + 20	
write(B)	

Not all concurrent schedules result in a consistent state. To demonstrate, consider the nonserial schedule in Table 5.6. After the execution of this schedule, the balances of accounts A and B become \$80 and \$180 respectively. Clearly, this is an inconsistent state since the sum of $A + B$ is not equal to \$300 any more.

In this schedule, the "write" operation of account A in transaction Tr_1 has been "delayed". During that time, Tr_2 transfers \$40 from account B to A, leading the temporary balances of A and B to be \$140 and \$160 respectively. At this point, the system is consistent. Then, when Tr_1 resumes to execute, it overwrites the value of account A, making it to be \$80. Tr_1 also obtains the new balance of B, and adds \$20 to it. So this schedule ends up with an inconsistent state with the final values of accounts A and B being $80 and $180 respectively.

Table 5.6: A Nonserializable Schedule

Tr_1	Tr_2
read(A)	
A = A - 20	
	read(B)
	B = B - 40
	write(B)
	read(A)
	A = A + 40
	write(A)
write(A)	
read(B)	
B = B + 20	
write(B)	

All serial schedules apparently hold the consistent constraints since the atomicity of each participant transaction is guaranteed. Needless to say, a schedule, after execution, must leave the database in a consistent state. In another word, the schedule should, in some sense, be equivalent to a serial schedule.

We say that a schedule is ***serializable*** if it produces the same outcome as a serial schedule. Schedules presented so far are all serializable except for the one in Table 5.6. Only by offering protection of serializability could a concurrent control protocol be correct.

5.2 Lock-Based Protocols

One way to ensure serializability is to require that access to data objects be done in a ***mutually exclusive manner***; that is, while one transaction accesses a data object, no other transaction can modify that object. The principal way to implement this is to allow a transaction to access a data object only if it is currently holding a **lock** on that object.

5.2.1 Locks

There are two modes of locks to a data object:

- **Shared**, also known as **optimistic**. If a transaction Tr_i has obtained a shared-mode lock (denoted by S) on object O, then Tr_i can read this object but it cannot write O.
- **Exclusive**, also known as **pessimistic**. If a transaction Tr_i has obtained an exclusive-mode lock (denoted by X) on object O, then Tr_i can both read and write O.

Given the above descriptions, Table 5.7 illustrates the relationship between these two modes of locks ($Tr_i \neq Tr_j$).

Table 5.7: The Relationship Between Locks

Tr_i requests	Tr_j holds	Tr_i acquires immediately
S	S	Yes
S	X	No
X	S	No
X	X	No

In order to access a data object, a transaction Tr_i must lock it first. If the object has not been locked yet, or locked by the same transaction, Tr_i could obtain the requesting lock mode immediately. If the object has already been locked by another transaction Tr_j, from Table 5.7, only in one situation could Tr_i acquire the desired lock instantly, i.e., the requesting mode is S, and the currently held mode by Tr_j is S. Under other circumstances, Tr_i must wait. A transaction may release a lock on a data object which it had locked at some earlier point. So if [illegible]ransaction obtains a share-mode lock on a data object, others can read but cannot write that data object. If a transaction issues an exclusive lock on a data object, others cannot read or write it.

We introduce the notations "lock-S(O)", "release-S(O)", "lock-X(O)", and "release-X(O)" to represent requesting a shared lock, releasing a shared lock, requesting an exclusive lock, and releasing an exclusive lock on the data object O respectively.

The example elaborated in Section 5.1 has to be extended with these new instructions so that the locking scheme would be mirrored in schedules. If we do not set up any restrictions on locks, the concurrent execution schedule in Table 5.8 is possible.

Table 5.8: A General Locking Schedule (which causes inconsistency)

Tr_1	Tr_2
lock-S(A)	
read(A)	
release-S(A)	
A = A - 20	
	lock-S(B)
	read(B)
	release-S(B)
	B = B - 40
	lock-X(B)
	write(B)
	release-X(B)
	lock-S(A)
	read(A)
	release-S(A)
	A = A + 40
	lock-X(A)
	write(A)
	release-X(A)
lock-X(A)	
write(A)	
release-X(A)	
lock-S(B)	
read(B)	
release-S(B)	
B = B + 20	
lock-X(B)	
write(B)	
release-X(B)	

In the above schedule, operations except for those locking instructions are in the same order as the schedule in Table 5.6. Thus, such a schedule brings the system to an inconsistent state with the values of A and B being \$80 and \$180 in the end, which violates serializability.

One potential reason for this is that both transactions should have only issued exclusive locks on data objects A and B since they change the values and eventually write them back. Another error-prone factor for the schedule is that one transaction releases the lock so early that another transaction may face an inconsistent state.

Suppose now that both transactions issue exclusive locks solely and releasing is delayed to the end of each transaction. The following schedule generates a new problem.

Table 5.9: A General Locking Schedule (which causes deadlock)

	Tr_1	Tr_2	
▷	lock-X(A)		
	read(A)		
	A = A - 20		
	write(A)		
		lock-X(B)	◁
		read(B)	
		B = B - 40	
		write(B)	
		lock-X(A)	waiting because of ▷
		read(A)	
		$A = A + 40$	
		write(A)	
		release-X(B)	◇
		release-X(A)	
waiting because of ◁	lock-X(B)		
	read(B)		
	$B = B + 20$		
	write(B)		
◇	*release-X(A)*		
	release-X(B)		

Since Tr_1 is holding an exclusive lock on A (marked ▷ in Table 5.9) and Tr_2 is requesting an exclusive-mode lock on A, Tr_2 is waiting for Tr_1 to release A. Likewise, since Tr_2 is locking account B in an exclusive mode (marked ◁) and

Tr_1 is requesting B, Tr_1 is waiting for Tr_2 to release B. As a consequence, we have arrived at a state where neither these transactions can ever proceed with its normal execution. This situation is called *deadlock*.

Deadlock is a state in which processes are blocked waiting for something that will never happen. Consider two processes, $Proc_1$ and $Proc_2$, that require resources A and B. If they attempt to acquire these resources in the opposite order, as shown in Table 5.10, then there is the potential for deadlock:

Table 5.10: A Sample Deadlock Situation

$Proc_1$:	get A	$Proc_2$:	get B
	get B		get A

If $Proc_1$ succeeds in acquiring resource A and $Proc_2$ succeeds in acquiring resource B, then they are deadlocked. In general there are four conditions that must be present before deadlock can occur:

1. Processes need to share resources that must be used under mutual exclusion.
2. Processes hold on to resources already allocated to them while waiting for more.
3. Resources are non-preemptible.
4. A circular chain of requests and allocations exists.

We can prevent deadlock by removing any of the four conditions from the system:

1. **Condition 1:** Don't allow process access to resources that must be used under mutual exclusion. A common example of this approach is having a spool directory for printing and putting the printing of documents into the heads of a printer daemon rather than letting processes print directly.
2. **Condition 2:** Make processes request all resources they require at once.

3. **Condition 3:** Add a mechanism for removing a resource from a process that has it. This can waste work though, because a process that is nearly finished may be required to relinquish a resource and start again later.

4. **Condition 4:** Order the resources and force processes to acquire the resources they require in order, making it impossible for a circular chain of requests and allocations to exist.

If deadlock cannot be avoided, a deadlock detection and recovery mechanism must be employed to find the deadlock and further recover from it.

Given the above discussion about deadlock, we can see that the *italic operations* in Table 5.9 can never get running because of the deadlock status. When deadlock occurs, one of the transactions must be rolled back so that the data objects that were once locked by that transaction are released. These data objects are then available to other transactions, which can continue with their executions.

Deadlock freedom is indispensable and critical for a concurrency control protocol since it is very dangerous for a system to enter a deadlock state. Even though a protocol does not ensure freedom from deadlock, it must supply a roll back mechanism if deadlock happens.

As we could learn from this section, the general locking protocol, which has no restrictions on locks, can guarantee neither serializability nor deadlock freedom. Naturally, we would impose a set of rules on how a transaction may lock and release data objects. We hope that the protocol ensures both serializability and freedom from deadlock when applying locks in a cautious manner.

5.2.2 The Two-Phase Locking Protocol

One popular locking protocol that ensures serializability is the *two-phase locking protocol*. It was first introduced by Eswaran, et al. [13]. This protocol requires that each transaction issue lock and release requests in two phases:

- **Growing phase.** A transaction may obtain locks but may not release any lock.

- **Shrinking phase**. A transaction may release locks but may not obtain any new locks.

A transaction starts within the growing phase initially. The transaction acquires locks as needed. Once the transaction begins to release a lock, it enters the shrinking phase and no more lock requests may be issued.

The two-phase locking protocol ensures serializability. Some formal proofs could be found in [25], [41]. However, it does not guarantee freedom from deadlock. To demonstrate this, observe that the schedule in Table 5.9 is a perfectly legal two-phase locking schedule. Both transactions get into the shrinking phase at the ◇ sign. As argued in Section 5.2.1, this schedule leads the system to a deadlock state.

As a rule, the two-phase locking protocol allows a transaction to lock a new data object only if it has not yet released any data object. In the absence of information concerning the manner in which data objects are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability [25]. Roll back mechanism needs to be appended when applying the two-phase locking protocol because the protocol does not ensure deadlock freedom.

5.2.3 Prothos' Optimistic Concurrency Control Mechanism

Before introducing Prothos' concurrency control scheme, let's consider Prothos' concurrency architecture first.

Figure 5.1 shows an example of three concurrent processes in Prothos. Each application corresponds to one POM object, and each POM object defines a transaction. All three processes connect to Avra RDBMS at the bottom.

There is not just one POM instance running as a central control of concurrent transactions. Instead, there may be many POM objects running at the same time in the system. So, the RDBMS layer becomes the only possible place to coordinate concurrent transactions. And that is why Prothos adopts the locking scheme to control concurrency.

Prothos uses an optimistic mechanism as its concurrency control scheme, which is an enhanced version of the two-phase locking protocol. The basic idea is that transactions issue optimistic locks most of the time. A transaction merely issues an exclusive-mode lock on a data object it needs to write back to the persistent storage when the transaction in the end enters the "commit" phase. The exclusive-lock requests indicate the start of "commit" phase.

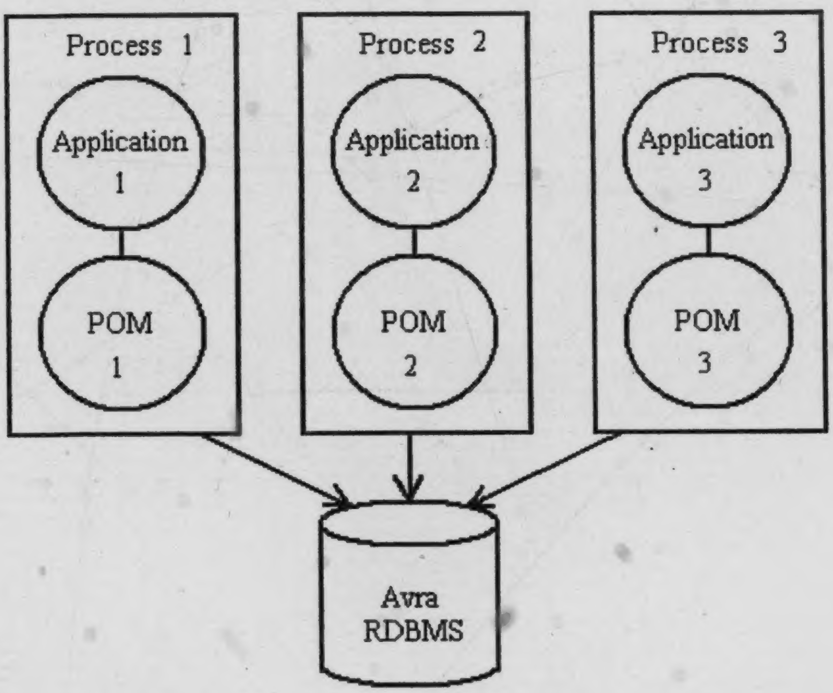

Figure 5.1: Concurrency Control in Prothos

Every transaction abides by the "growing" and "shrinking" rules defined in the two-phase locking protocol. However, the semantics of lock modes introduced in Section 5.2.1 have been modified.

If a transaction Tr_i has obtained a shared-mode lock on object O, then Tr_i can read and write this object. At the same time, other transactions may read O and possibly write O. The transaction must issue the exclusive lock on O first in order to write O. If the transaction has held a shared lock on O already, it will *upgrade* the shared-mode lock to exclusive-mode lock. And when the exclusive lock is released, the shared lock is also gone. If a transaction Tr_i has obtained an exclusive-mode lock, then Tr_i can both read and write O. And no other transaction can read or write it. The point here is that for any data object, if there is an exclusive lock on it, there can only be one transaction holding that lock at a time in the system.

Given the above descriptions of POM's locks, Table 5.11 illustrates the relationship between these two modes of locks ($Tr_i \neq Tr_j$).

Table 5.11: The Relationship Between Locks in POM

Tr_i requests	Tr_j holds	Tr_i acquires immediately
S	S	Yes
S	X	No
X	S	**Yes**
X	X	No

Compared to Table 5.7, the only difference here is emphasized by the boldface **Yes**. That is the requesting transaction can obtain the exclusive lock immediately even though the data object is currently shared-locked by another transaction. This makes the Prothos' concurrency control mechanism less strict. Intuitively, it leads to more throughput than the two-phase locking protocol does. We will show an example later on to demonstrate this assertion is true.

Notice that we use all lock-related notations henceforth to represent the locks defined in Prothos, not the ones in Section 5.2.1.

Prothos implements its optimistic concurrency control scheme in the following way:

- Every data object has an intrinsic attribute called "_Delta" as we discussed in Chapter 3. The *persistent object manager* (POM) defines this attribute for every database schema in Prothos. "_Delta" has the type of integer, and is hidden from the user. The value of "_Delta" is always increasing for the purpose of identifying changes to the data object. For the data object O, we use Δ_O to denote the value of "_Delta" in O.

- If a transaction wants to read a data object during its execution, it requests a shared-mode lock. If the object is at present locked exclusively by another transaction, the requesting transaction has to wait. A timeout is set up for the waiting transaction to avoid an arbitrarily long inactivity. When the timeout expires, the requesting transaction has to abort.

- When a transaction successfully issues a shared-mode lock on an object, it is able to read the object. In Prothos, the transaction reads not only the values of actual attributes, but also the value of the attribute "_Delta" in the object. These two reads must happen atomically. For each transaction, Prothos locally keeps the most recent value of "_Delta" of each data object the transaction accesses.

- After a write-transaction finishes performing all its basic operations, it needs to write all involved data objects back. The transaction requests the exclusive-mode lock to every participant object. If any of the objects is currently holding an exclusive lock issued by another transaction, the requesting transaction has no choice but to wait. Similarly, a timeout is established in order to prevent the requesting transaction from waiting forever.

- When a transaction successfully issues the exclusive-mode locks on all its participant objects, it enters the "commit" phase. Before it actually writes back all objects, it compares its local copy of each object's "_Delta" value to the one retrieved from the database. If all local copies of "_Delta" are equal to the latest ones from the database, it means no other transactions have ever changed these data objects, and the transaction can safely write the objects back. At the same time, each object's "_Delta" value is increased to mark this transaction's completion. Otherwise, it aborts since some objects have been modified during its execution.

- While a transaction holds an exclusive-mode lock on a data object, no other transaction can read or write it. All other locking requests have to wait till the exclusive lock is released. If a transaction aborts or successfully commits, it releases locks on all invloving data objects.

To conform to the optimistic locking protocol in Prothos, the two transactions developed in the example in Section 5.1, Tr_1 and Tr_2, are now adjusted to what is shown in Table 5.12 and Table 5.13 respectively.

Table 5.12: Transaction Tr_1 in Prothos

Tr_1 :	lock-S(A)
	read(A and Δ_A)
	A = A - 20
	lock-S(B)
	read(B and Δ_B)
	B = B + 20
	lock-X(A and B)
	compare(Δ_A and Δ_B)
	write(A and Δ_A)
	write(B and Δ_B)
	release-X(A and B)

Table 5.13: Transaction Tr_2 in Prothos

Tr_2 :	lock-S(B)
	read(B and Δ_B)
	B = B - 40
	lock-S(A)
	read(A and Δ_A)
	A = A + 40
	lock-X(A and B)
	compare(Δ_A and Δ_B)
	write(A and Δ_A)
	write(B and Δ_B)
	release-X(A and B)

As we mentioned before, Prothos' concurrency control mechanism is less strict than the two-phase locking protocol, which leads to more transaction throughput. We now show an example to demonstrate this point.

The schedule in Table 5.14 is the modified version of the schedule in Table 5.9. The operations in this schedule follow the Prothos' concurrency control scheme we have just introduced.

In this schedule, Tr_2 can successfully commit whereas Tr_1 has to abort. When Tr_2 requests the exclusive locks on A and B, it will obtain the locks immediately even though A has already been shared-locked by Tr_1. When Tr_2 writes A back, it also increases the value of Δ_A so that the comparison of Δ_A and Δ_B in Tr_1 will fail, which forces Tr_1 to abort.

Table 5.14: A Schedule under Prothos' Concurrency Control Scheme

	Tr_1	Tr_2	
	lock-S(A)		
	read(A and Δ_A)		
	A = A - 20		
		lock-S(B)	
		read(B and Δ_B)	
		B = B - 40	
		lock-S(A)	
		read(A and Δ_A)	
		A = A + 40	
		lock-X(A and B)	will NOT wait
		compare(Δ_A and Δ_B)	
		write(A and Δ_A)	
		write(B and Δ_B)	
		release-X(A and B)	
		commits	
	lock-S(B)		
	read(B and Δ_B)		
	B = B + 20		
	lock-X(A and B)		
will fail since Δ_A has been changed by Tr_2	compare(Δ_A and Δ_B)		
	release-X(A and B)		
	aborts		

This schedule protects serializability. What's more, compared to the similar two-phase schedule in Table 5.9, this schedule does not generate the deadlock problem. The fact that Tr_2 successfully commits in this schedule demonstrates the assertion: Prothos' concurrency control scheme leads to more transaction throughput than the two-phase locking protocol does.

There are several reasons why Prothos uses this optimistic locking scheme to guard serializability and deadlock freedom. First of all, there might be enormous not-read-only transactions running concurrently. The exclusive locking time should be limited as much as possible so as for other transactions to be able to read data objects. The optimistic locking mechanism only issues the exclusive-mode locks when a transaction tries to enter the "commit" phase.

Moreover, the best scenario for the user is that there is no wait, and no rollback, either. However, in order to ensure both serializability and freedom from deadlock, one is inevitable. Since Prothos suggests short-duration transactions, minimum user-interaction transactions, and low-volume transactions, the current waiting and rollback schemes are satisfactorily valid.

There is a very important but implicit assumption in Prothos' optimistic locking scheme. That is, when a transaction enters into the "commit" phase, it would finish up in a certain period of time. In reality, the current POM module does not implement this assumption. As a result, there might be a **"starvation"** problem in Prothos which we will be investigating in Section 5.3.

5.2.4 Other Protocols

There are a lot more concurrency control schemes to protect serializability. All these do so by either delaying an operation or aborting the transaction that issued the operation. The most common ones are locking protocols, timestamp-ordering schemes, validation techniques, and multiversion schemes. We have already studied several locking protocols in great detail. In this section, we briefly review other schemes.

- **Timestamp-ordering schemes.** A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction Tr_i is smaller than the timestamp of transaction Tr_j, then the scheme ensures that the produced schedule is equivalent to a serial schedule in which Tr_i appears before Tr_j. This is done by rolling back a transaction whenever such an order is violated. More information about **timestamp-ordering schemes** can be found in [5], [36].

- **Validation techniques**. A validation scheme is an appropriate concurrency control method in cases where the majority of transactions are read-only transactions, and thus the rate of conflicts among these transactions may be very low. A unique fixed timestamp is associated with each transaction in the system. The serializability order is determined by the timestamp of the transaction. A transaction in this scheme is never delayed. It must, however, pass a validation test, in order to complete. If it does not pass the validation test, it is rolled back to its initial state. More information about **validation techniques** can be found in [3], [26].

- **Multiversion schemes**. A multiversion concurrency control scheme assumes that each write(O) operation creates a new version of data object O. When a read(O) operation is issued, the system selects one of the versions of O to be read. The concurrency control scheme ensures that the selection of the version to be read is done in a manner that ensures serializability. This is accomplished through the use of timestamps. A read operation always succeeds, while a write operation may result in the rollback of the transaction. More information about **multiversion schemes** can be found in [6], [41].

5.3 Model Checking Lock-Based Protocols

In Section 4.3.2, we addressed that Prothos' non-concurrent transaction model is too easily understandable to bother the model checking technique. The lock-based concurrency control protocols, on the other hand, are not straightforward to comprehend because even for a given number of transactions, there might exist a great deal of valid schedules under certain protocols. We could take full advantage of model checking to reason behavior of different kinds of protocols.

It is worthwhile to employ the model checking technique to understand those lock-based protocols since the concurrency control is probably the most important property of transaction management in Prothos.

5.3.1 Model Construction

In Section 5.2, we discussed two most significant features of a concurrency control scheme — serializability and deadlock freedom, and analyzed how different protocols ensure them. The models we build in this section would certainly center around these two key properties. Instead of arguing and exemplifying, we develop several formal models to demonstrate behavior of those protocols.

Figure 5.2 shows the state transition diagram of Prothos' optimistic locking concurrency control mechanism in the current implementation.

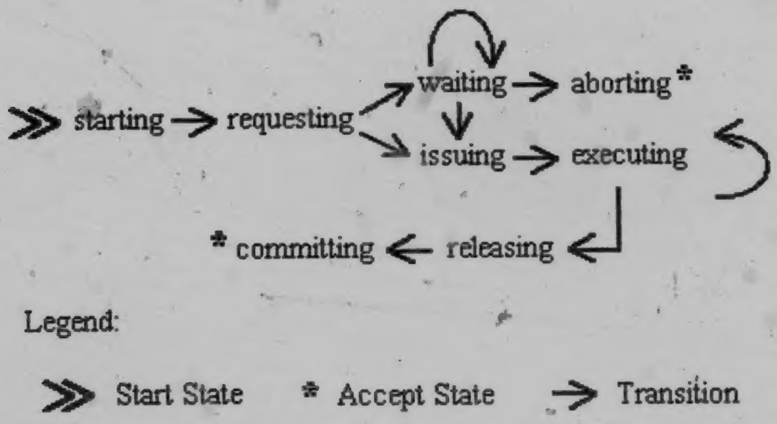

Figure 5.2: State Transition of Concurrency Control in Prothos (current implementation)

The state transition diagram in Figure 5.2 is captured from the current implementation of POM[1]. The code is the only "formalism" we can get.

Appendix A shows the SMV model built for checking the concurrency behavior of POM. The SMV file is self-explained, but here are some points we want to emphasize:

- The current implementation uses "per table locking", which means the locking object is a database table, not a specific data object. For each table, there is one "exclusive" transaction, one "waiting" transaction, and a set of "shared" transactions to record this table's locking status. In the SMV model, we use the variable "ca_w" and "ca_x" to represent the waiting transaction and exclusive-locking transaction, and use the array "ca_s" to represent the set of shared-locking transactions.

- The capitalized "Delta_A" represents the actual _Delta value in the database, so it is defined as a global variable in the SMV model. Each transaction keeps a local copy of "Delta_A" in "delta_A". When the transaction successfully committs, it increases the value of the global "Delta_A". The same rules can apply to "Delta_B".

- The global boolean variable "semaphore" in the SMV model is captured by the "/locked" directory creation in the implementation. If a transaction wants to obtain the exclusive lock on a table, it tries to create the "/locked" directory in the file system. Since the directory creation is mutually exclusive in UNIX/LINUX operating system, this guarantees that at most one process(transaction) can get the semaphore successfully and then enter into the critical section.

- The "waiting_timeout" defined in the SMV model is of type integer. This reflects the waiting timeout defined in the code. Currently, the waiting timeout is 20 seconds. So we use the range [0,20] as the scope of the "waiting_timeout" variable in the SMV model.

---

[1] Version 4.1.3.10, released on March 3, 2003. Author: James Konkin. © Copyright 2003 Avra Software Lab, Inc.

Given the above correspondences, we can focus on the SMV model to analyze and check the behavior of POM.

Every transaction starts in the "starting" state, and immediately goes to "requesting" state. If the transaction can get the locks for each data object, it enters to "issuing" state; otherwise, it gets into "waiting" state. When the transaction is waiting, a timeout is set for it. When the waiting timeout is expired, and the transaction still cannot get all the locks, it enters the "aborting" state, and the transaction is rolled back. If it can get all the locks during waiting, it will move to the "issuing" state.

The next state of "issuing" is "executing", in which the transaction does committing execution. It takes some time to finish the committing execution, so there is a transition loop defined in state "executing". When the executing is finished, the transaction releases all the locks, and commits. The "aborting" and "committing" states are accepting states in this model.

Note that finite-state limitations of the logic prevent us from building an SMV model that accepts arbitrary number of transactions. In the context of concurrency control protocols, it is rational to start reasoning with two interacted transactions. If the protocol is incorrect, problems would be disclosed in this two-transaction model for certain. If the protocol passes this test, from the architectural perspective, we could view these two transactions as a new "abstract" transaction. Since the "abstract" transaction holds serializability and freedom from deadlock inside and presents atomicity outside, it could be combined with another atomic transaction to form a new model. This new model would pass the test without doubt and be regarded as an "abstract" transaction again. In some cases, it might be possible to model check the inductive step to further verify the correctness of the protocol over arbitrary number of transactions.

5.3.2 Specifications

If we consider the two concurrent write-transaction situation, the example constructed in Section 5.1 is still suitable. The consistency constraint is that the sum, $A + B$, should always be \$300, which is used to check serializability.

For each transaction, if it eventually goes into "committing" or "aborting" state, it is not deadlocked. In another word, if a transaction makes progress, it is not deadlocked.

$$AG((A\_value\ +\ B\_value) = 300) \tag{5.1}$$

$$\begin{aligned}
&AG\ (\ (tr1.state = starting) \rightarrow \\
&(AF\ (tr1.state = committing \mid tr1.state = aborting)\ )\ ) \\
&\&\ AG(\ (tr2.state = starting) \rightarrow \\
&(AF\ (tr2.state = committing \mid tr2.state = aborting)\ )\ )
\end{aligned} \tag{5.2}$$

Spec 5.1 is the serializability specification and **Spec 5.2** is the deadlock specification. Both specifications are **false** in the general locking protocol model, which correctly validates that the non-restriction locking protocol could not guarantee either serializability or freedom from deadlock. The VW model checker[2] provides us a counter example every time the model fails to satisfy the specification.

The two-phase locking protocol model passes the serializability test with the value of **Spec 5.1** being **true**. **Spec 5.2** turns out to be **false** in this model, which confirms the announcement that the two-phase locking protocol may guide the system to a deadlock state.

We have gained some experience on writing specifications:

1. It is related to model construction.

   In our case, we cannot arbitrarily choose transaction's semantics since we want to check serializability. For example, the **Spec 5.1** does not allow us to define a transaction that withdraws $35 from an account. The specifications indeed render the model construction in some sense.

2. Check whether the model itself is correct – $AG\ 0$.

---

[2] Author: Ken McMillan. © Copyright 2002 Cadence Design Systems, Inc.

Check firstly whether the model is correct by including the false specification (AG 0). If this specification is true, the model itself is wrong, and any other specification will be true. This model cannot check anything because it does not exist at all.

3. Make it false to check.

   Sometimes, it is necessary to write down "false" specifications. These specifications can perform as debuggers. They can help us find out what's wrong in the model. Exposing mistakes is also one of the advantages of formal methods [21].

4. Keep old "wrong" specifications for a later-version check.

   We find out that it is important to record the plausible specifications that were tried and rejected. These records will help improve the quality of specifications, guide future versions of the model, and prevent the repetition of unfruitful work.

5.3.3 Starvation

The SMV model we built based on Figure 5.2 satisfies **Spec 5.1**. But **Spec 5.2** turns out to be false in this case. This is not caused by deadlock though. The reason that **Spec 5.2** fails is the "starvation" problem as we mentioned in Section 5.2.3. Starvation is also known as the "indefinite postponement" situation.

One major assumption in Prothos' optimistic locking concurrency control scheme is that any transaction would come to an end when it went into the "commit" phase. If there were no such an assumption, the system would possibly get into the *starvation* state, that is, one process keeps seizing the execution thread and never halts; meanwhile, other processes are incapable of running because their desired resources are exclusively occupied by the "live" process. As a result, those transactions inevitably fall into an infinite loop of requesting, waiting, aborting, and requesting. This is apparently an undesirable situation since the system is not making progress at all.

In this case, **Spec 5.2** cannot be true. Although the transaction of being starved does go from "starting" to "aborting", the transaction that causes this starvation will stay in "executing" forever. Table 5.15 gives part of the counter example generated by the VW model checker. The trace shows why **Spec 5.2** is false in the model based on Figure 5.2.

Table 5.15: Part of the Counter Example

	running process	**state**
loop starts →	Tr_2	executing
	Tr_1	starting
	Tr_1	requesting
	Tr_1	waiting
	Tr_1	aborting
	Tr_1	starting
loop ends →	Tr_2	executing

So **Spec 5.2** not only checks freedom from deadlock, but also tests absence of starvation. To solve the starvation problem, we introduce another timeout[3] and build the future improvement model which is shown in Figure 5.3. When a transaction enters the "executing" state, the system automatically establishes a timeout for it. If the timeout expires and the transaction has not finished execution yet, this transaction is forced to roll back, and all locks on data objects currently held by it are released. The dollar sign in Figure 5.3 shows this new-added transition.

It is possible to set up such an executing timeout for transactions in Prothos to prevent the starvation problem. This could be a new feature for the new version of POM. We could configure an approximate running time for transactions since Prothos is designed for light-weight web applications. The SMV model built based on Figure 5.3 satisfies both **Spec 5.1** and **Spec 5.2**, which in effect verifies the correctness of the proposed solution.

---

[3] The first timeout set in the model is to guard deadlock freedom. If a transaction is waiting for another transaction to release the exclusive-mode lock on certain data object, a timeout is set up for it in order to avoid an arbitrarily long inactivity, see Section 5.2.3.

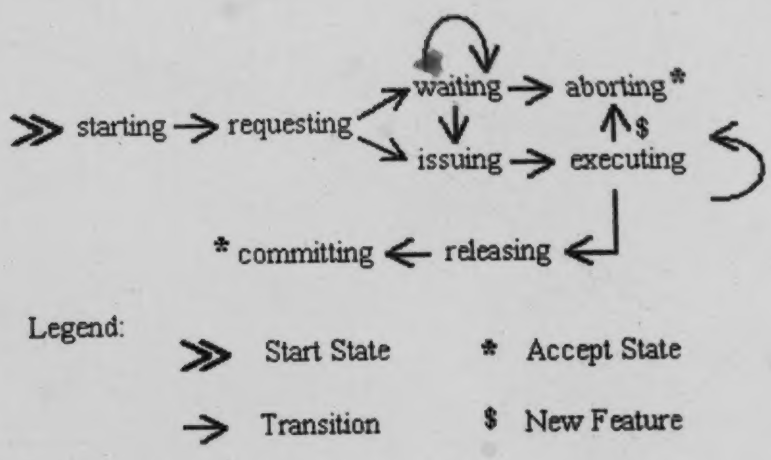
starting
requesting
waiting
aborting *
issuing
executing
$
* committing
releasing
Legend:
Start State
* Accept State
Transition
$ New Feature

Figure 5.3: State Transition of Concurrency Control in Prothos (future improvement)

5.3.4 Hermes

We have actually experienced an application built on top of Prothos that suffered from the starvation problem. In the fall 2002 academic term [23], a course project called "Hermes"[4] was developed using Prothos framework.

This project aims to deliver a database-style email client module for applications built on top of the Prothos framework. A user will be able to download his or her email messages from any standard account to be stored within the application's database. Multiple views of messages, including sorting, filtering and grouping, will be available to and configurable by the user to suit his or her workflow. The handling of attachments (sending, receiving and viewing) is key in any model of group workflow and will therefore be fully supported by the module.

It sometimes takes a long period of time for Hermes to retrieve email messages from different servers such as pop.mail.yahoo.com, imap.srv.ualberta.ca, and so forth. This long-duration transaction is possibly caused by the large number of email messages, the bulky size of attachments, or even the terrible traffic jam on the Internet.

---

[4] Authors: Tyler Botten, Matt Edwards, Francis Huber, Doug Hurst, Trina Magera, Ryan Warden, and Yifan Xu. © Copyright 2002 – 2003 Department of Computing Science, University of Alberta.

Since Prothos' POM does not take the starvation prevention into consideration, the transaction locks quite a few data objects and then moves into the "executing" phase. Due to several reasons, this transaction could take arbitrarily long and lead the system to the starvation state. Whenever this occurs, we have to manually remove the locks in the database in order for Hermes to resume running. This is obviously not the right way to use a framework. On one hand, such a "heavy-weight" application does not fit in Prothos' specific domain. On the other hand, the starvation defect discovered by model checking does affect the application.

We see from the above example that model checking helps present a critical observation. As a rule, the model checking technique plays a key role in analyzing the concurrency control protocols. If model checking (or some other formal methods) were used during Prothos' design phase, the starvation problem should have disappeared.

Although Wing [47] argued that formal methods can be applied in all phases of system development, as far as software architecture is concerned, we advise people apply formal methods to some vital parts in the software architecture **in the design phase**, or at least, earlier than the coding stage.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we have presented a formal way to understand the transaction management component's behavior in a software framework. The software component has been formally studied through three different layers. During this procedure, we argued that formal methods can indeed help improve software design and development. And as far as software architecture is concerned, they should be employed in the design phase of the whole software lifecycle.

We studied the software component's architecture first. Our methodology was to investigate several data models and then build up POM's ontology. The properties revealed in this level laid the groundwork for transaction-level behavior's discussion in the software architecture.

Then, we explored the atomicity of the single transaction model in the framework. The software component protected the database consistency both physically and logically in this second layer. The utilization of model checking in this context did not exhibit many exciting results.

The concurrency control scheme was of extreme importance for the transaction management component. Detailed descriptions of several lock-based protocols were provided. We took full advantage of model checking to illustrate and validate two most significant features of a concurrency control protocol — serializability and deadlock freedom. The starvation problem was discovered in the software framework by applying formal methods. We thus concluded our work in light of the question whether and how formalism can help software architecture.

The main contributions of this thesis are summarized as follows:

- Formalization draws out the "real" architecture. This is because by applying formal methods, we must study the system and pay attention to details. We demonstrate that "layering" is a good architectural principle.

- The goal of POM is to manipulate data objects as in object-oriented programs but they are actually kept around as in relational databases. The ontology provides the consistent terminology throughout the software framework and applications and among the stakeholders.

- As far as the single transaction model was concerned, we claimed that it was so easily understandable that the model checking technique did not fit in this context.

- Model checking played an important role in our work in checking the critical behavior of the software component: the original designs were correctly tested, and some problems and suggestions were pointed out.

- We put forward one potential improvement for the current version of Prothos. That is, in order to solve the starvation problem in the optimistic locking protocol, an approximate transaction finish timeout can be set.

- An SMV model was exploited in order to reason the key properties in the software framework. We offered some valuable recommendations on how to write specifications in SMV. We also inductively proved the correctness of our model from the architectural point of view.

- Our work guides the formal testing in the transaction management field in some sense. Serializability, deadlock freedom, and starvation freedom are necessary behaviors that need to be checked under a concurrency-supportive transaction management circumstance.

- "Formal methods" is still not the silver bullet. In order to understand, analyze, or reason a system, it is better to have a prototype, some sample code, or a concrete example.

Finally, we are able to answer the question — whether and how formalism can help software architecture. Based on this work, our answer is that, once again, formal methods can indeed help improve software design and development. And as far as software architecture is concerned, they should be employed in the design phase of the whole software lifecycle.

6.2 Future Work

While our work presents a complete formal way to comprehend the transaction management component's behavior in a software framework, there are still a number of areas for possible future research.

First of all, how to generate sufficient test case(s) is still an open issue. The test case depends on which feature in the architecture we want to check. For example, the current version of POM uses per table locking. For every table in the database, there is one "exclusive" transaction, one "waiting" transaction, and a set of "shared" transactions to record this table's locking status. In this case, two concurrent write-transaction case is not sufficient. In order to capture the locking implementation, we build a model of three write transactions and two read-only ones to test the specifications. This model turns out to be necessary, but still not sufficient. In our practice, we only employ base test cases. It will be beneficial to find out how to generate sufficient test case(s), given the "state space explosion" problem of model checking.

Moreover, it will be interesting and worthwhile to find out the appropriate formal methods to better illustrate the software component's properties in the first two levels — the basic database operational level and the non-concurrent transaction level. Our work mainly concentrates on the model checking technique, which we choose as the representative of formal methods. As we showed in Chapter 3 and 4, model checking is not the proper tool to some extent under these circumstances.

In addition, developing a collection of formal models of concurrency control protocols for the software component will greatly improve Prothos' adaptability. Besides lock-based protocols, as we introduced in Section 5.2.4, there are numerous schemes to guard serializability. Every protocol has its advantages and disadvantages. Model checking other schemes will soundly increase the value of the software component by making it easier to be adapted to other configurations.

Furthermore, there are several other inviting features in the concurrent transaction layer in Prothos that deserve further investigation, such as nested transactions, cascading rollback, etc. It will be absolutely meaningful to extend our models to check these behaviors. Since we pay most our attention to SMV models in particular, selecting other formal languages and tool sets will be another reasonable way to follow.

Last but not least, from the architectural standpoint, capturing the overall behavior of the entire software framework will definitely be necessary and attractive. In this thesis, we have explored the applicability of SMV models in analyzing the transaction management component's behavior. Making good use of formal methods across a range of components will be of benefit to ultimately comprehend the behavior of the whole framework. Without a doubt, this kind of work will shed further light on how to better answer the question — ***whether and how formalism can help software architecture***.

Bibliography

[1] S. Alagic. *Relational Database Technology*. Springer-Verlag New York, Inc., 1986.

[2] R. J. Allen. A formal approach to software architecture. Ph.D. Thesis, Technical Report Number: CMU-CS-97-144, Carnegie Mellon University, 1997.

[3] M. Bassiouni. Single-site and distributed optimistic protocols for concurrency control. In *IEEE Transactions on Software Engineering, Volume SE-14, Number 8 (August 1988), pages 1071–1080*, 1988.

[4] D. Begay. The eurotri case, some aspects of using formal methods in an industrial project. Internal Report, Techinical Report Number: 1141-96, LaBRI, 1996.

[5] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the International Conference on Very Large Data Bases (1980), pages 285–300*, 1980.

[6] P. A. Bernstein, N. Goodman, and M. Y. Lai. Analyzing concurrency control when user and system operations differ. In *IEEE Transactions on Software Engineering, Volume SE-9, Number 3 (May 1983), pages 233–239*, 1983.

[7] A. Bouhoula. Automated theorem proving by test set induction. In *Journal of Symbolic Computation, 23(1): pages 47–77*, 1997.

[8] J. P. Burgess. Basic tense logic. In *Handbook of Philosophical Logic. Volume 2: Extensions of Classical Logic, pages 89–134*, 1984.

[9] R. W. Butler, V. A. Carreno, B. L. Di Vito, K. J. Hayhurst, and P. S. Miner. Nasa langley's research and technology-transfer program in formal methods. In *Proceedings of the Tenth Annual Conference on Computer Assurance (1995), pages 135–150*, 1995.

[10] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, volume 131 of Lecture Notes in Computer Science, Springer-Verlag*, 1981.

[11] F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. In *IEEE Transactions on Software Engineering, Volume SE-2, Number 2 (June 1976), pages 80–86*, 1976.

[12] A. Descartes and T. Bunce. *Programming the Perl DBI.* O'Reilly & Associates, Inc., 2000.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. In *Communications of the ACM, Volume 19, Number 11 (November 1976), pages 624–633*, 1976.

[14] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the International Conference on Software Engineering (1997), pages 491–501*, 1997.

[15] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. *Designing Object-Oriented Frameworks.* CRC Handbook of Object Technology, CRC Press, 1998.

[16] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of ACM SIGSOFT (1994): Foundations of Software Engineering, pages 175–188*, 1994.

[17] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON (1997), pages 169–183*, 1997.

[18] R. Greenlaw and H. J. Hoover. *Fundamentals of the Theory of Computation, Principles and Practice.* Morgan Kaufmann Publishers, Inc., 1998.

[19] Object Management Group. The common object request broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

[20] T. R. Gruber. A translation approach to portable ontology specifications. In *Knowledge Acquisition, Volume 5, Number 2, pages 199–220*, 1993.

[21] A. Hall. Seven myths of formal methods. In *IEEE Software, Volume 6, Number 9 (September 1990), pages 11–19*, 1990.

[22] R. Hayes and R. D. Schlichting. Facilitating mixed language programming in distributed systems. In *IEEE Transactions on Software Engineering, Volume SE-13, Number 12 (December 1987), pages 54–64*, 1987.

[23] H. J. Hoover. Cmput 401: Software process and product management. In *Department of Computing Science, University of Alberta*, Fall term, 2002.

[24] D. Hou and H. J. Hoover. Towards specifying constraints for object-oriented frameworks. In *Journal of Information Systems Frontiers, 4(4): pages 393–407*, 2002.

[25] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, Inc., 1991.

[26] H. T. Kung and J. T. Robinson. Optimistic concurrency control. In *ACM Transactions on Database Systems, Volume 6, Number 2 (June 1981), pages 312–326*, 1981.

[27] D. A. Lamb. Idl: Sharing intermediate representations. In *ACM Transactions on Programming Languages and Systems, Volume 9, Number 3 (July 1987), pages 297–318*, 1987.

[28] D. C. Luckham and J. Vera. An event-based architecture definition language. In *IEEE Transactions on Software Engineering, Volume SE-21, Number 9 (September 1995), pages 717–734*, 1995.

[29] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of ACM SIGSOFT (1996): Sourth Symposium Foundations of Software Engineering, pages 3–14*, 1996.

[30] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[31] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering, Volume SE-26, Number 1 (January 2000), pages 70–93*, 2000.

[32] T. H. Merrett. *Relational Information Systems.* Reston Publishing Company, Inc., 1984.

[33] C. J. Nix and B. P. Collins. The use of software engineering, including the z notation, in the development of cics. In *Quality Assurance, September 1988: pages 103–110*, 1988.

[34] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software Engineering Notes, 17(4): pages 40–52*, 1992.

[35] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. In *Journal of Systems and Software, 6(4): pages 307–334*, 1986.

[36] D. Reed. Implementing atomic actions on decentralized data. In *ACM Transactions on Computer Systems, Volume 1, Number 1 (February 1983), pages 3–23*, 1983.

[37] Microsoft Research. Software model checking project at microsoft research. http://research.microsoft.com/slam.

[38] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[39] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. In *IEEE Transactions on Software Engineering, Volume SE-21, Number 4 (April 1995), pages 314–335*, 1995.

[40] M. Shaw and D. Garlan. *Software Architecture – Perspectives on an Emerging Discipline.* Prentice Hall, Inc., 1996.

[41] A. Silberschatz. A multi-version concurrency control scheme with no rollbacks. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (1982), pages 216–223*, 1982.

[42] Avar Software. Prothos reference document on peoria. http://peoria.cs.ualberta.ca/~fraser/prothos/prothos.htm.

[43] D. Szafron. Cmput 625: Object-oriented programming languages. In *Department of Computing Science, University of Alberta*, Fall term, 2002.

[44] W. F. Tichy. Software development control based on module interconnection. In *Proceedings of the International Conference on Software Engineering (1979), pages 29–41*, 1979.

[45] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Inc., 1982.

[46] P. Valduriez and G. Gardarin. *Analysis and Comparison of Relational Database Systems.* Springer-Verlag New York, Inc., 1989.

[47] J. M. Wing. A specifier's introduction to formal methods. In *IEEE Computer, Volume 23, Number 9 (September 1990), pages 8–24*, 1990.

Appendix A

The SMV Model

```
------------------------------------------------------------
------------------------------------------------------------
-- Subject: Model checking the locking schema in Prothos' POM (v 4.1.3.10)
-- Note: Give semantics to each transaction to check serializability
--        deadlock freedom, and starvation freedom
-- Author: Nan Niu (nan@cs.ualberta.ca)
-- Create Date: 17:46, April 24, 2003
-- Last Modified Date: 05:30, Aug 08, 2003
-- Last Modified By: Nan Niu (nan@cs.ualberta.ca)
------------------------------------------------------------
------------------------------------------------------------

-- Model checking the locking schema in Prothos, Version 4.1.3.10
-- released date: March 5, 2003
-- Code mainly focused on DB.pm
-- In this model, there are two transactions: tr1, tr2
-- Two DB tables, and two DB records accordingly: A and B
-- with initial value of 100 and 200 respectively
-- As we're using per table locking
-- And A and B are the only record considered in each table
-- We use A_s, A_w, and A_x instead of table names
-- The relationship is:

-- tr1: transferring 20 from A to B
-- tr2: transferring 40 from B to A

-- Note: the 'w' (waiting) lock is set automatically;
-- No transaction requests a lock mode 'w' at all.

MODULE main

VAR

    -- GLOBAL variables defined here
    -- the directory '/locked' is the semaphore in the locking schema
    semaphore : boolean;
    - - the capitalized 'Delta_A' and 'Delta_B' are actual values in the
database
    - - since there are only two write-transactions, the maximum value of the
global delta will be 2
    Delta_A : 0..2;
    Delta_B : 0..2;

    A_value : 0..500;
    B_value : 0..500;

    A_s : array 1..2 of {tr1, tr2, nil};
    A_w : {tr1, tr2, nil};
    A_x : {tr1, tr2, nil};

    B_s : array 1..2 of {tr1, tr2, nil};
    B_w : {tr1, tr2, nil};
    B_x : {tr1, tr2, nil};

    - - two transactions defined as follows
    - - the instance variable like 'state'
    - - 'local_A' are defined inside the module

    tran1 : process transaction(tr1, semaphore, Delta_A, Delta_B, A_value,
B_value, A_s, A_w, A_x, B_s, B_w, B_x);
    tran2 : process transaction(tr2, semaphore, Delta_A, Delta_B, A_value,
B_value, A_s, A_w, A_x, B_s, B_w, B_x);


ASSIGN

    init(semaphore) := 0;

    init(Delta_A) := 0;
    init(Delta_B) := 0;

    init(A_value) := 100;
    init(B_value) := 200;

    init(A_s[1]) := nil;
    init(A_s[2]) := nil;
    init(A_w) := nil;
    init(A_x) := nil;

    init(B_s[1]) := nil;
    init(B_s[2]) := nil;
    init(B_w) := nil;
    init(B_x) := nil;

- - Specifications

- - check whether the model itself is correct
```

```
SPEC
    AG 0


- - check semaphore

SPEC
    AG ( (tran1.state=issuing | tran1.state=executing |
tran1.state=releasing) → semaphore )


SPEC
    AG ( (tran2.state=issuing | tran2.state=executing |
tran2.state=releasing) → semaphore )


- - check the waiting_timeout

SPEC
    AG ( (tran2.state=requesting) → (tran2.waiting_timeout=0) )

SPEC
    AG ( (tran1.state=waiting) → !(tran1.waiting_timeout=0) )


- - check serializability

SPEC
    AG (A_value + B_value = 300)

- - check deadlock freedom and starvation freedom

SPEC
    AG ( (tran1.state=starting) → AF (tran1.state=committing |
tran1.state=aborting) )

SPEC
    AG ( (tran2.state=starting) → AF (tran2.state=committing |
tran2.state=aborting) )


- - make the specification "false" to give us a counter example

SPEC
    AG !(tran1.waiting_timeout=17)


- - A transaction has 8 state: starting, requesting,
- - waiting, issuing, aborting, executing, releasing, committing

- - each transaction will keep a local copy of delta_A and delta_B
- - in order to perform the optimistic checking
- - about the waiting timeout
- - a waiting timeout is 20 means the transaction starts waiting;
- - when the timeout is decreased to 1, it stops waiting, and rolls back;
- - and if the value is 0, the transaction is not waiting at all

- - about the executing timeout
- - an executing timeout is 10 means the transaction starts executing;
- - when the timeout is decreased to 1, it stops executing,
- - and goes to aborting;
- - and if the value if 0, the transaction is not executing at all


MODULE transaction(id, semaphore, A_value, B_value, A_s, A_w, A_x, B_s,
B_w, B_x)


VAR
    state : {starting, requesting, waiting, issuing, aborting, executing,
aborting, committing};

    delta_A : 0..2;
    delta_B : 0..2;

    waiting_timeout : 0..20;

    executing_timeout : 0..10;


ASSIGN
    init(state) := starting;
    init(delta_A) := 0;
    init(delta_B) := 0;
    init(waiting_timeout) := 0;
    init(executing_timeout) := 0;


    - - The following annotations are important
    - - They give the reader the basic idea of model construction


    - - TRANSITION 001 : starting → requesting

    - - CASE 001 (TRANSITION 001)
    - - affects: state only


    - - TRANSITION 002 : requesting → issuing

    - - CASE 002 (TRANSITION 002)
    - - you can get the semaphore, both A_x and B_x are 'nil'
    - - A_w is either 'nil' or 'id'; so is B_w
```

```
-- affects: state, semaphore, A_x, B_x


-- TRANSITION 003 : requesting → waiting

-- CASE 003 (TRANSITION 003)
-- you're in state 'requesting', but cannot get semaphore
-- only check whether you should set A_w and B_w or not
-- affects: state, waiting_timeout, A_w, B_w

-- CASE 004 (TRANSITION 003)
-- you're in state 'requesting', and you can get the semaphore
-- however, you just cannot get the lock you want
-- affects: state, waiting_timeout, A_w, B_w


-- TRANSITION 004 : waiting → issuing

-- CASE 005 (TRANSITION 004)
-- pretty much like CASE 002
-- you're in the state 'waiting', and you get the semaphore
-- every locking condition is good for you
-- affects: state, waiting_timeout, semaphore, A_w, A_x, B_w, B_x


-- TRANSITION 005 : waiting → waiting

-- CASE 006 (TRANSITION 005)
-- pretty much like CASE 003
-- you're in state 'waiting', but cannot get semaphore
-- at the same time, your waiting_timeout is NOT 1,
-- and is NOT 0, either
-- should check A_w and B_w
-- affects: state, waiting_timeout, A_w, B_w

-- CASE 007 (TRANSITION 005)
-- pretty much like CASE 004
-- you're in state 'waiting', and you can get the semaphore
-- at the same time, your waiting_timeout is NOT 1,
-- and is NOT 0, either
-- however, you just cannot get the lock you want
-- affects: state, waiting_timeout, A_w, B_w


-- TRANSITION 006 : waiting → aborting

-- CASE 008 (TRANSITION 006)
-- pretty much like CASE 006
-- you're in state waiting', but cannot get semaphore
-- at the same time, your waiting_timeout is 1, don't wait any more
-- go to 'aborting'
-- should check A_w and B_w
-- affects: state, waiting_timeout, A_w, B_w

-- CASE 009 (TRANSITION 006)
```

```
- - pretty much like CASE 007
- - you're in state 'waiting', and you can get the semaphore
- - however, you cannot get the lock you want
- - at the same time, your waiting_timeout is 1, don't wait any more
- - go to 'aborting'
- - affects: state, waiting_timeout, A_w, B_w


- - TRANSITION 007 : issuing → executing

- - CASE 010 (TRANSITION 007)
- - the new feature in POM – executing timeout
- - affects: state, executing_timeout


- - TRANSITION 008 : executing → aborting

- - CASE 011 (TRANSITION 008)
- - you're in state 'executing',
- - and the executing_timeout is 1, don't execute any more,
- - go to 'aborting'
- - affects: state, semaphore, A_x, B_x, executing_timeout


- - TRANSITION 009 : executing → executing
- - TRANSITION 010 : executing → releasing

- - CASE 012 (TRANSITION 009 & 010)
- - you're in state 'executing',
- - you can go either 'executing' or 'releasing' arbitrarily
- - given CASE 011 defines first, that is, the executing_timeout is not 1
- - affects: state

- - CASE 013 (TRANSITION 009)
- - you're in state 'executing',
- - you have to check the executing_timeout
- - that is, if the executing_timeout is not 1 or not 0,
- - decreasing the timeout
- - affects: executing_timeout

- - CASE 014 (TRANSITION 010)
- - you're in state 'releasing',
- - you have to make sure the executing_timeout is 0
- - affects: executing_timeout


- - TRANSITION 011 : releasing → committing

- - CASE 015 (TRANSITION 011)
- - you're in state 'releasing', you must go to 'committing' next
- - affects: state, semaphore, A_value, B_value, A_x, B_x


next(state) := case
```

```
        - - CASE 001
        state=starting : requesting;

        - - CASE 002
        state=requesting & !semaphore & (A_x=nil & B_x=nil) & (
(A_w=nil | A_w=id) & (B_w=nil | B_w=id) ) : issuing;

        - - CASE 003
        state=requesting & semaphore : waiting;

        - - CASE 004
        state=requesting & !semaphore & ( !(A_x=nil) | !(B_x=nil) ) :
waiting;
        state=requesting & !semaphore & (A_x=nil & !(A_w=nil) &
!(A_w=id) ) : waiting;
        state=requesting & !semaphore & (B_x=nil & !(B_w=nil) &
!(B_w=id) ) : waiting;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : issuing;

        - - CASE 006
        state=waiting & semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) : waiting;

        - - CASE 007
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & ( !(A_x=nil) | !(B_x=nil) ) : waiting;
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & (A_x=nil & !(A_w=nil) & !(A_w=id) ) : waiting;
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & (B_x=nil & !(B_w=nil) & !(B_w=id) ) : waiting;

        - - CASE 008
        state=waiting & semaphore & waiting_timeout=1 : aborting;

        - - CASE 009
        state=waiting & !semaphore & waiting_timeout=1 & ( !(A_x=nil) |
!(B_x=nil) ) : aborting;
        state=waiting & !semaphore & waiting_timeout=1 & (A_x=nil &
!(A_w=nil) & !(A_w=id) ) : aborting;
        state=waiting & !semaphore & waiting_timeout=1 & (B_x=nil &
!(B_w=nil) & !(B_w=id) ) : aborting;

        - - CASE 010
        state=issuing : executing;

        - - CASE 011
        state=executing & executing_timeout=1 : aborting;

        - - CASE 012
        state=executing : {executing, releasing};

        - - CASE 015
        state=releasing & (delta_A=Delta_A & delta_B=Delta_B) :
committing;

        - - otherwise, stay unchanged
        1 : state;

    esac;


    next(delta_A) := case

        - - CASE 015
        state=releasing & (delta_A=Delta_A & delta_B=Delta_B) : delta_A
+ 1;
    esac;


    next(delta_B) := case

        - - CASE 015
        state=releasing & (delta_A=Delta_A & delta_B=Delta_B) : delta_B
+ 1;
    esac;


    next(waiting_timeout) := case

        - - CASE 003
        state=requesting & semaphore : 20;

        - - CASE 004
        state=requesting & !semaphore & ( !(A_x=nil) | !(B_x=nil) ) : 20;
        state=requesting & !semaphore & (A_x=nil & !(A_w=nil) &
!(A_w=id) ) : 20;
        state=requesting & !semaphore & (B_x=nil & !(B_w=nil) &
!(B_w=id) ) : 20;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : 0;

        - - CASE 006
        state=waiting & semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) : waiting_timeout - 1;

        - - CASE 007
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & ( !(A_x=nil) | !(B_x=nil) ) : waiting_timeout - 1;
```

```
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & (A_x=nil & !(A_w=nil) & !(A_w=id) ) :
waiting_timeout - 1;
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & (B_x=nil & !(B_w=nil) & !(B_w=id) ) :
waiting_timeout - 1;

        - - CASE 008
        state=waiting & semaphore & waiting_timeout=1 : 0;

        - - CASE 009
        state=waiting & !semaphore & waiting_timeout=1 & ( !(A_x=n[illegible] |
!(B_x=nil) ) : 0;
        state=waiting & !semaphore & waiting_timeout=1 & (A_x=nil &
!(A_w=nil) & !(A_w=id) ) : 0;
        state=waiting & !semaphore & waiting_timeout=1 & (B_x=nil &
!(B_w=nil) & !(B_w=id) ) : 0;

        - - otherwise, stay unchanged
        1 : waiting_timeout;

    esac;


    next(executing_timeout) := case

        - - CASE 010
        state=issuing : 10;

        - - CASE 011
        state=executing & executing_timeout=1 : 0;

        - - CASE 013
        state=executing & !(executing_timeout=1) &
!(executing_timeout=0) : executing_timeout - 1;

        - - CASE 014
        state=releasing : 0;
        - - otherwise, stay unchanged
        1 : executing_timeout;

    esac;


    next(semaphore) := case

        - - CASE 002
        state=requesting & !semaphore & (A_x=nil & B_x=nil) & (
(A_w=nil | A_w=id) & (B_w=nil | B_w=id) ) : 1;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : 1;

        - - CASE 011
        state=executing & executing_timeout=1 : 0;

        - - CASE 015
        state=releasing : 0;

        - - otherwise, stay unchanged
        1 : semaphore;

    esac;


    next(A_value) := case

        - - avoid A_value out of range
        A_value > 460 : A_value;
        A_value < 20 : A_value;

        - - CASE 011
        state=releasing & id=tr1 : A_value - 20;
        state=releasing & id=tr2 : A_value + 40;

        - - otherwise, stay unchanged
        1 : A_value;

    esac;


    next(B_value) := case

        - - avoid B_value out of range
        B_value > 480 : B_value;
        B_value < 40 : B_value;

        - - CASE 011
        state=releasing & id=tr1 : B_value + 20;
        state=releasing & id=tr2 : B_value - 40;

        - - otherwise, stay unchanged
        1 : B_value;

    esac;


    next(A_s[1]) := case

        - - otherwise, stay unchanged
        1 : A_s[1];
    esac;


    next(A_s[2]) := case

            - - otherwise, stay unchanged
            1 : A_s[2];

    esac;


    next(A_w) := case

            - - CASE 002
            state=requesting & !semaphore & (A_x=nil & B_x=nil) & A_w=id :
nil;

            - - CASE 003
            state=requesting & semaphore & A_w=nil : id;

            - - CASE 004
            state=requesting & !semaphore & ( !(A_x=nil) & A_w=nil ) : id;

            - - CASE 005
            state=waiting & !semaphore & (A_x=nil & B_x=nil) & A_w=id :
nil;

            - - CASE 006
            state=waiting & semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & A_w=nil : id;

            - - CASE 007
            state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & ( !(A_x=nil) & A_w=id ) : nil;

            - - CASE 008
            state=waiting & semaphore & waiting_timeout=1 & A_w=id : nil;

            - - CASE 009
            state=waiting & !semaphore & waiting_timeout=1 & A_w=id : nil;

            - - otherwise, stay unchanged
            1 : A_w;

    esac;


    next(A_x) := case

            - - CASE 002
            state=requesting & !semaphore & (A_x=nil & B_x=nil) & (
(A_w=nil | A_w=id) & (B_w=nil | B_w=id) ) : id;
```

```
        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : id;

        - - CASE 011
        state=executing & executing_timeout=1 & A_x=id : nil;

        - - CASE 015
        state=releasing & (A_x=id | A_x=nil) : nil;

        - - otherwise, stay unchanged
        1 : A_x;

    esac;


    next(B_s[1]) := case

        - - otherwise, stay unchanged
        1 : B_s[1];

    esac;


    next(B_s[2]) := case

        - - otherwise, stay unchanged
        1 : B_s[2];

    esac;


    next(B_w) := case

        - - CASE 002
        state=requesting & !semaphore & (A_x=nil & B_x=nil) & B_w=id :
nil;

        - - CASE 003
        state=requesting & semaphore & B_w=nil : id;

        - - CASE 004
        state=requesting & !semaphore & ( !(B_x=nil) & B_w=nil ) : id;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & B_w=id :
nil;

        - - CASE 006
        state=waiting & semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & B_w=nil : id;
```

```
        - - CASE 007
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & ( !(B_x=nil) & B_w=id ) : nil;

        - - CASE 008
        state=waiting & semaphore & waiting_timeout=1 & B_w=id : nil;

        - - CASE 009
        state=waiting & !semaphore & waiting_timeout=1 & B_w=id : nil;

        - - otherwise, stay unchanged
        1 : B_w;

    esac;


    next(B_x) := case

        - - CASE 002
        state=requesting & !semaphore & (A_x=nil & B_x=nil) & (
(A_w=nil | A_w=id) & (B_w=nil | B_w=id) ) : id;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : id;

        - - CASE 011
        state=executing & executing_timeout=1 & B_x=id : nil;

        - - CASE 015
        state=releasing & (A_x=id | A_x=nil) : nil;

        - - otherwise, stay unchanged
        1 : B_x;

    esac;


FAIRNESS
    running
```

University of Alberta

FORMALLY UNDERSTANDING THE BEHAVIOR OF A FRAMEWORK'S TRANSACTION MANAGEMENT

by

©

Nan Niu

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2003

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisisitons et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

Our file *Notre référence*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

Canadä

University of Alberta

Library Release Form

Name of Author: Nan Niu

Title of Thesis: Formally Understanding the Behavior of a Framework's Transaction Management

Degree: Master of Science

Year this Degree Granted: 2003

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Date: Sep 5, 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Formally Understanding the Behavior of a Framework's Transaction Management** submitted by Nan Niu in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. H. James Hoover
Supervisor

Dr. James Miller

Dr. Ken Wong

Date: Sep 2, 2003

Abstract

As the size and complexity of software systems increase, the overall system structure, which is also known as *software architecture*, becomes increasingly critical in software design and development. *Formal methods* provide mathematically based techniques that are particularly useful for systematically specifying and verifying computer systems. This thesis is a case study on leveraging formal methods to Prothos, a domain-specific database-centric web application framework. It is motivated in light of the question — **whether and how formalism can help software architecture**. Both structual and behavioral properties of the transaction management component in the framework are studied. Formal approaches based on model checking techniques are presented. Our practice verifies the current implementation, clarifies implicit assumptions, and provides potential improvements for the software framework. Also, our work can help develop a systematic way to tackle an architectural study.

To my mother and father with love and gratitude.

Acknowledgements

I have been lucky to have had opportunities to work with many brilliant people during my course study and thesis research. This has definitely helped to shape my thesis in ways I would never have been able to achieve on my own.

First and foremost, I would like to thank my supervisor, Jim Hoover, not only for his scholarly guidance and keen editorial eyes, but also for his encouragement and support throughout the work. His patience of going into gory details, his tolerance of my wayward requests, and above all, his willingness to contribute insightful comments to my work, have all been instrumental in making this thesis a reality.

This work could not have been done without the generous assistance of James Konkin at Avra Software. His authoritative, detailed, and prompt responses have solved many puzzles about Prothos. I am especially grateful to Matt Edwards for informative conversations and help about Prothos.

I wish to express my sincere gratitude to my committee: Dr. James Miller and Dr. Ken Wong. The final version of this thesis has greatly benefited from their comments. I owe a great debt of gratitude to Dr. Miller whose expertise and help in software verification and model evolution has been a valuable contribution to my work.

Thanks also go to SERL members: Dr. Eleni Stroulia, Dr. Paul Sorenson, Daqing Hou, Changyu Yin, Julian Fogel, Pablo Figueroa, and John Shillington for their discussion with respect to my work. Changyu proofreaded the early version of the thesis. Her comments greatly improved my presentation and eleminated a number of writing errors.

I also appreciate the advice of Dr. Steve Easterbrook, Dr. Amy Felty, Dr. Osmar Zaiane, Dr. Joerg Sander, Dr. Davood Rafiei, and Dr. Mario Nascimento. Many thanks to the Department of Computing Science and Jim for offering me the financial assistantship during my study.

Last, but certainly not least, I would like to thank my parents for their never-ending and unwavering love, care, support, and understanding which have clung to me all my life. It is on their behalf that this thesis is dedicated. I am similarly grateful to other members in our family: maternal grandmother, paternal grandmother, maternal uncle, maternal aunt, and my lovely maternal cousin. All my families have been waiting so long for this moment to come true; I am glad that their waiting has finally been rewarded. They are what makes it all worthwhile!

Contents

1 Introduction 1
1.1 Motivation 1
1.2 Thesis Structure 2

2 Background Practice and Related Research 4
2.1 DBMS 4
2.1.1 RDBMS 6
2.1.2 OODBMS 6
2.2 Software Framework 7
2.2.1 Software Architecture 9
2.2.2 Prothos 9
2.2.3 POM Structure 12
2.3 Formal Methods 14
2.3.1 Model Checking 15
2.4 Related Work 18
2.5 Thesis Scope 20

3 POM Architecture 21
3.1 Data Models 21
3.1.1 Object-Oriented Data Model 21
3.1.2 Relational Data Model 22
3.1.3 POM Data Model 24
3.2 Ontology 26
3.2.1 POM::Object 26
3.2.2 POM::BusinessClass 28
3.2.3 POM::Session 28
3.2.4 Relationships 29
3.3 Patterns of Use 29
3.4 Data Integrity Issues 31
3.4.1 Domain Constraints 32
3.4.2 Referential Integrity 32
3.4.3 Functional Dependencies 33

4 Transaction Model in Prothos 34
4.1 Data Object Message Simplification 34
4.2 The Storage Hierarchy 35
4.3 The Transaction Model 36
4.3.1 The State Diagram 36
4.3.2 Model Checking The Transaction Model 39

5 Concurrency Control 42
5.1 Schedules 42
5.2 Lock-Based Protocols 46
5.2.1 Locks 47
5.2.2 The Two-Phase Locking Protocol 51
5.2.3 Prothos' Optimistic Concurrency Control Mechanism . 52
5.2.4 Other Protocols 58
5.3 Model Checking Lock-Based Protocols 59
5.3.1 Model Construction 60
5.3.2 Specifications 62
5.3.3 Starvation 64
5.3.4 Hermes 66

6 Conclusions and Future Work 68
6.1 Conclusions 68
6.2 Future Work 70

Bibliography 72

A The SMV Model 76

List of Tables

3.1 Mapping from POM Data Model to Relational Data Model . . 26

4.1 The Transaction Model's Transition Table 38

5.1 Transaction Tr_1 43
5.2 Transaction Tr_2 43
5.3 A Serial Schedule (in which Tr_1 is followed by Tr_2) 44
5.4 A Serial Schedule (in which Tr_2 is followed by Tr_1) 44
5.5 A Serializable Schedule 45
5.6 A Nonserializable Schedule 46
5.7 The Relationship Between Locks 47
5.8 A General Locking Schedule (which causes inconsistency) . . . 48
5.9 A General Locking Schedule (which causes deadlock) 49
5.10 A Sample Deadlock Situation 50
5.11 The Relationship Between Locks in POM 54
5.12 Transaction Tr_1 in Prothos 56
5.13 Transaction Tr_2 in Prothos 56
5.14 A Schedule under Prothos' Concurrency Control Scheme . . . 57
5.15 Part of the Counter Example 65

List of Figures

1.1 Study of POM in Three Layers 3

2.1 Levels of Abstraction in a Database 5
2.2 Application Developed from a Framework (adopted from [15]) 8
2.3 Prothos' Architecture 11
2.4 POM's Structure 13
2.5 Data Flow through DBI 14

3.1 A Sample Class Diagram 22
3.2 A Sample Instance Diagram 23
3.3 Class Diagram of Relational Data Model 24
3.4 Class Diagram of POM Data Model 25

4.1 The Storage Hierarchy 35
4.2 The Transaction State Diagram 37
4.3 The CTL Diagram of The Transaction Model 40

5.1 Concurrency Control in Prothos 53
5.2 State Transition of Concurrency Control in Prothos (current implementation) 60
5.3 State Transition of Concurrency Control in Prothos (future improvement) 66

Chapter 1

Introduction

1.1 Motivation

As the size and complexity of software systems increase, the overall system structure, which is also known as *software architecture*, becomes increasingly critical in software design and development.

Formal methods have a sound mathematical basis, typically given by a formal specification language. They are particularly useful for systematically describing and reasoning about computer-based systems.

Thus, an interesting question in day-to-day software engineering is that: ***whether and how formalism can help software architecture***.

To find out solutions to such a problem, we look at both structural and behavioral aspects of software architecture. Since architecture and design are broad areas, we have focused on domain-specific application frameworks. These kinds of frameworks are key to modern application development. Frameworks promote reuse and rapid development by constraining the space of possible solutions. Formal methods in constrained situations have a greater chance of being useful. Also, investment in reusable frameworks can justify investment in formal methods.

Our work centers around a software framework called "Prothos" used to create database-centric web applications. The framework is fairly mature since it has been developed, used, and evolved for a couple of years, and a number of applications have been built on top of it.

The whole software framework is a complex system. The transaction management component, named *persistent object manager* (POM), plays an important role in Prothos. It not only allows the user to define, create, and manipulate persistent business objects, but also provides transaction services to user's applications. Without POM, each application developer would have to build their own mechanism for object persistence (leading to interoperability problems between applications) and would also have to make decisions about whether to allow concurrent access and how to implement this.

It is indispensable and significant to capture the behavior of POM in Prothos. Our approach is to analyze, model, and reason about the core features in a formal way. The ultimate goal is to verify the correctness of the original design, clarify the crucial but implicit assumptions in the framework, and provide potential improvements for the current version of Prothos. Our practice is to investigate the following questions, given a mature software framework:

- Can formal methods really help us understand software architecture?
- And when should we make use of formal methods in the software lifecycle?

Our case study is to check whether this approach is worth doing, and to learn something about the problems—both formal methods and software architecture. From this experience, we could then develop a more systematic approach to further similar practice. Our experience can also provide a "road map" of how to tackle an architectural study.

1.2 Thesis Structure

The next chapter surveys related work of this thesis. We provide the background overview of our work: database management systems, software framework, and formal methods. We also review some related research and practice in the literature.

Chapters 3 to 5, as a group, present a formal way to understand the transaction management component's behavior in Prothos. The software component is formally studied through three different layers. Each chapter focuses on characteristics in one specific level. The hierarchy is shown in Figure 1.1.

Chapter 5	concurrency control
Chapter 4	single transaction model
Chapter 3	POM architecture

Figure 1.1: Study of POM in Three Layers

POM's architecture is considered in Chapter 3. We analyze several data models including the underlying POM data model. Then, we build up POM's ontology to provide the shared knowledge throughout systems and among stakeholders. Patterns of use are provided and the goal is emphasized. In particular, the issues of data integrity are addressed. The POM architecture builds up the foundations for discussion in the other two levels.

We explore the non-concurrent transaction model in Prothos in Chapter 4. Atomicity is probed in this layer. Model checking is applied in this context. This level lays the underpinning for the concurrency control of transactions in the framework.

Chapter 5 concentrates on concurrency issues. Several lock-based concurrency control protocols are introduced and formal models are developed in order to check the most considerable properties in this layer — serializability and deadlock freedom. This is a key chapter as it presents Prothos' optimistic concurrency control mechanism as well as detects the starvation problem in the framework.

Finally, we conclude in Chapter 6 with a summary of contributions and a speculation of future work.

Chapter 2

Background Practice and Related Research

2.1 DBMS

In any type of organization, considerable resources and activities are dedicated to the gathering, filing, processing, and exchanging of data based on well-established procedures in order to achieve specific goals. For example, in a university, data management systems are set up for the purpose of providing education services, whereas in a bank, data organization is based on the provision of financial services. The development of data management technology with the implementation of database systems tries to meet this need.

A *database management system* (DBMS) consists of a collection of interrelated data and a set of programs to access that data. The collection of data, usually referred to as the *database*, contains information about one particular enterprise. The primary goal of a DBMS is to provide an environment that is both *convenient* and *efficient* to use in retrieving and storing database information [25].

The objective of database systems is to manage large bodies of information. The management involves both the definition of structures and the provision of mechanisms for data manipulation. In addition, a DBMS must consider other factors such as security, distribution, concurrency, integrity, and so on.

One of the major functions a database system should provide is an abstract view of the data. It is obvious that between the computer, dealing with bits, and the user dealing with applications, there exist some levels of abstractions. A fairly standard viewpoint regarding abstractions in a database is depicted in Figure 2.1.

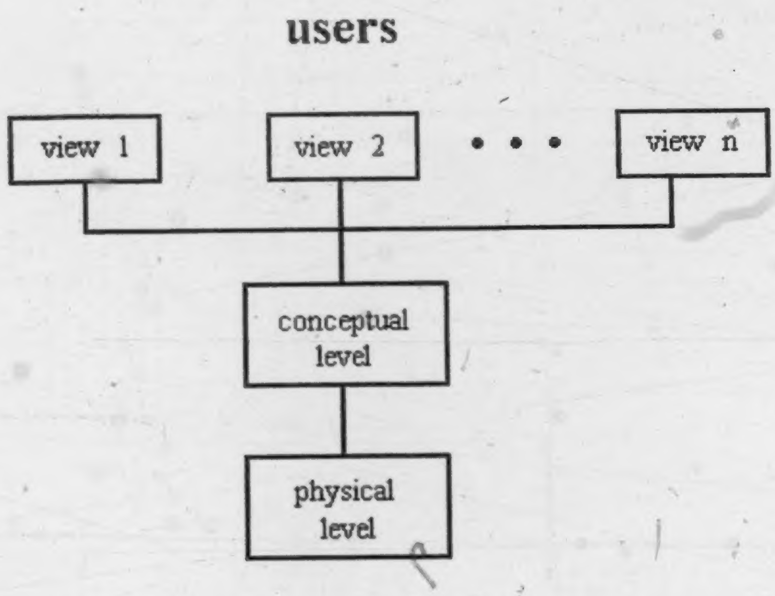

Figure 2.1: Levels of Abstraction in a Database

- The *physical* level of abstraction describes *how* the data are actually stored. The physical database resides permanently on secondary storage devices, such as disks and tapes [45].

- The *conceptual* abstraction level is responsible for defining *what* data are indeed stored in the database, and the relationships that exist among the data. Data stored in this level could be thought of as a description of enterprise information that is implementation-independent.

- A *view* is an abstract model of a portion of the entire database. This level of abstraction is necessary since users of the database system may only be concerned with part of the information.

2.1.1 RDBMS

The *relational database management system* (RDBMS) adopts the relational data model as its basis. The mathematical concept underlying the relational model is the set-theoretic *relation*, which is a subset of the Cartesian product of a list of domains.

A relational database consists of a collection of relations. The members of a relation are called *tuples*. It helps to view a relation as a *table*, where each row in a table is a tuple and each column corresponds to an *attribute*.

A relational schema is a description of the structure of relations in a relational database. The schema is a relatively static concept which defines the logical design of the database. Basically, a relation schema is a list of attributes and their corresponding domains.

RDBMS have been extensively studied in computer science, and there are many references in the literature like [1], [32], [46]. We assume that the reader has a general knowledge of RDBMS. In Chapter 3, we will discuss Prothos' data model, and further investigate the properties of RDBMS.

2.1.2 OODBMS

The *object-oriented database management system* (OODBMS) integrates database technology with the object-oriented programming paradigm.

The object-oriented data model is based on the concept of encapsulating data and code that operates on that data into a single unit called an *object*. The interface between an object and the rest of the system is defined by a set of *messages*.

An object could be abstractly viewed as a triplet: identity, state, and behavior. An object retains its identity even if its state changes over time. Its behavioral features are exhibited by responding to messages sent through the interface.

Similarly structured objects are grouped into *classes*. The set of classes is structured into a sub- and superclasses hierarchy using the concept of *inheritance*.

There are some notable differences between the relational model and the object-oriented one. Above all, complex objects cannot be modelled using a single table in relational models, but instead require additional tables and foreign keys. A complex object has a collection[1] of instance variables. The value of an instance variable can be an object or a collection of objects. This characteristic enables the complex object to have an arbitrarily complicated internal structure. In object-oriented models, the collection is an attribute of the object.

Moreover, the relational model does not provide the notion of inheritance. Additionally, there are no mechanisms for associating operations with the definition of data objects in the relational schema, and the behavioral semantics of objects is dispersed in application programs. Finally, the concept of the identity of objects does not apply to tuples of a relational database. We cannot have two identical tuples in a relation, but can have two distinct objects with equal attributes except the identity. That is, the object identity always provides a unique key.

2.2 Software Framework

A *software framework* is a reusable design and implementation for all or part of a software system. An object-oriented framework is typically implemented as a set of abstract classes, which define the core functionality of the framework along with concrete classes for specific applications.

Frameworks exploit the commonalities in design and implementation between a family of related applications. Software engineers use frameworks to reduce the cost of building complex applications.

Users of the framework complete or extend the framework by adding custom application specific components or functions to produce an application [15]. A good software framework consists of three parts: software architecture, implementation, and hooks. Each hook provides a concise solution to a focused requirement or problem within the context of the framework [14].

---

[1] A collection includes any container such as sets, lists, classes, etc.

The concepts associated with frameworks can be identified in Figure 2.2. Applications are developed on top of frameworks by filling in missing pieces and customizing the framework in the appropriate areas.

The core of the framework normally consists of abstract classes that define the generic structure and behavior of the framework. Framework libraries are extensions to the framework core comprising concrete components that can be used with little or no modification by applications. Classes in the unused library are not required by the application. Application extensions define specific classes that do not exist in the framework.

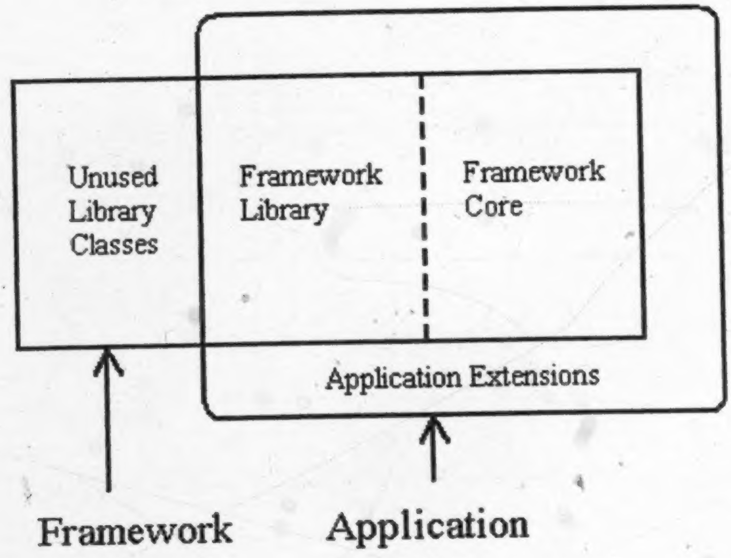

Figure 2.2: Application Developed from a Framework (adopted from [15])

Frameworks are meant to be reused to develop applications. Application-specific functional requirements are usually implemented by applications. Some non-functional properties like reusability, extensibility, and flexibility, then become dominant criteria when evaluating a framework.

2.2.1 Software Architecture

All software systems, including frameworks, must have a solid foundation at the level of *software architecture.* Software architecture studies the higher level issues involved in software design, such as the organization of a system as a composition of components, global control structures, physical distribution of subsystems, scaling and performance, selection among design alternatives, and so forth.

Two popular definitions of software architecture are proposed by Perry and Wolf [34] and Shaw and Garlan [40]. Both of them could be abstractly viewed as a triplet. In [34], software architecture is interpreted as a model that consists of: *elements*, *form*, and *rationale.* In [40], software architecture involves *components*, *connectors*, and *configurations.* No matter which definition we choose, software architecture provides us a model of the large scale structural and behavioral properties of systems.

Software architecture raises the level of abstraction at which developers can reason about their systems. A system's architecture provides a model of the system that suppresses implementation details and increases the independence of system components, permitting many issues to be localized. By suppressing these details at the architectural level, the architect can concentrate on the analyses and decisions that are most crucial to the system structure [2].

2.2.2 Prothos

Prothos is a domain-specific application framework used in creation of database-centric web applications. It is designed and developed by Avra Software Lab Inc[2].

Web applications are multi-user, with access control based on the logged-in users. However, the *Hyper Text Transfer Protocol* (HTTP) is connectionless, therefore Prothos provides full support for server-side session and page instance data. With a secure login and session management scheme, access to pages, workflow, and database records are controlled.

---

[2] More information could be found at: http://www.avrasoft.com .

Web applications support multiple concurrent transactions, and Prothos provides the optimistic concurrency control mechanism to guarantee serializability and deadlock freedom. This prevents concurrent users from taking actions based on obsolete data, such as placing an order that has been already modified by someone else during the placing time.

Since Prothos web applications are transaction based, if a failure is going to occur, it will occur safely. Prothos uses a structured exception handling mechanism. This ensures that if anything goes wrong during the processing of a transaction, no changes are made to the database and the user is informed of the failure as well as it being logged along with the relevant debug information.

Prothos applications use a three-tier architecture, in which the user interface functions, business and workflow functions, and the database management functions are delivered by relatively independent pieces of software. This separation of functionality makes it much easier to maintain web applications developed with Prothos.

The Prothos framework is written in Perl, a programming language which is available on all Microsoft and Unix-like back ends. Perl includes its own facilities for interfacing with almost any relational database management system, with full support for atomic transactions and referential integrity.

Figure 2.3 illustrates the Prothos' architecture. The acronyms and abbreviations are explained as follows:

• **UA**	User Agent (an instance of a user working with the application)
• **UI**	User Interface
• **BC**	Business Class
• **POM**	Persistent Object Manager
• **UIM**	User Interface Manager
• **SQL**	Structured Query Language

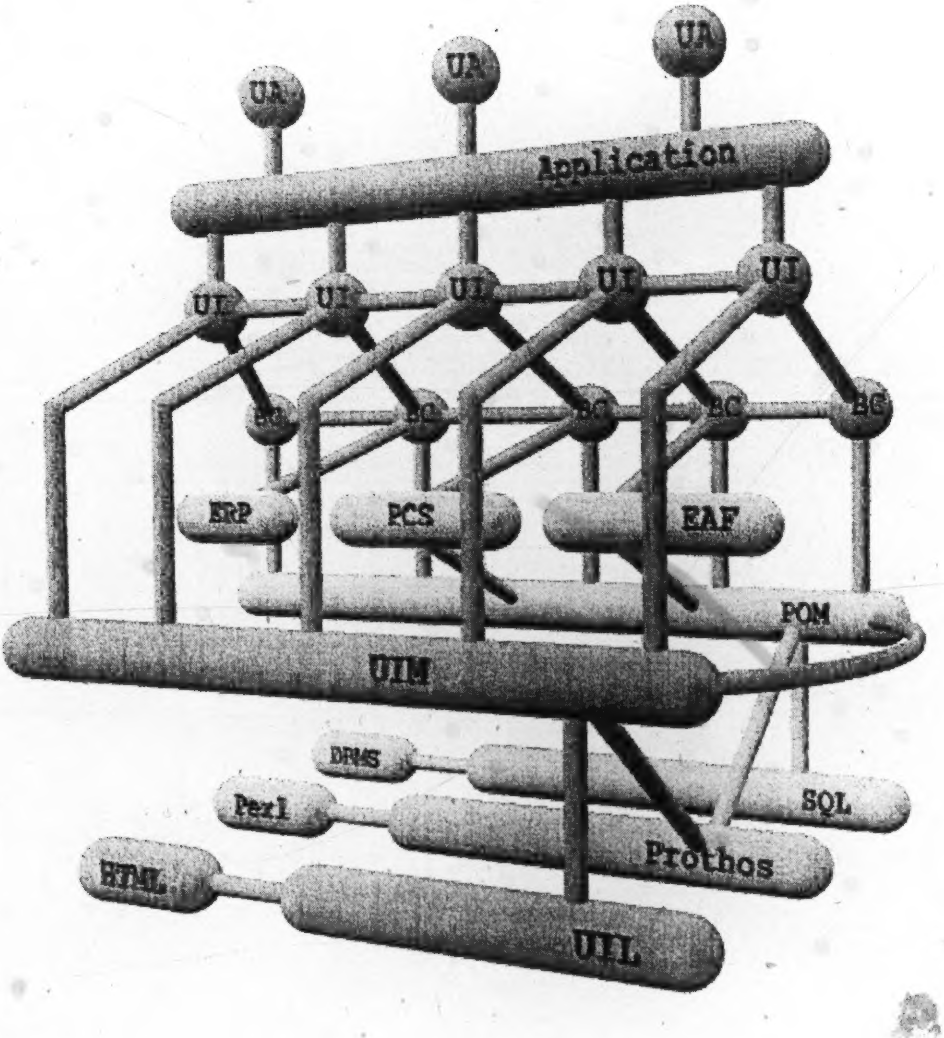
UA
UA
UA
Application
UI
UI
UI
UI
UI
BC
BC
BC
BC
BC
ERP
PCS
EAF
POM
UIM
DBMS
SQL
Perl
Prothos
HTML
UIL

Figure 2.3: Prothos' Architecture

• **DBMS**	Database Management System
• **Perl**	Practical Extraction and Reporting Language (*also known as:* Pathologically Eclectic Rubbish Lister)
• **UIL**	User Interface Language
• **HTML**	Hyper Text Markup Language
• **EAF***	Engineering Application Framework
• **PCS***	Product Catalog Service
• **ERP***	Enterprise Resource Planning

The components marked with asterisk signs are examples of services that one could attach to Prothos using the service hooks as required by an application. Strictly speaking, they are not part of Prothos framework.

There are many web applications built from Prothos like the *Software Engineering Research Laboratory* (SERL)[3] at the Department of Computing Science, University of Alberta, the *Alberta Software Engineering Research Consortium* (ASERC)[4], etc.

Prothos was successfully introduced to a one-term senior course to help students develop their projects [23]. The fact that all five teams[5] finished their web applications on time and in high quality demonstrates the reusability, robustness, and maturity of Prothos.

2.2.3 POM Structure

As shown in Figure 2.3, Prothos is a layered system. The whole architecture is sufficiently complicated that it is sometimes unrealistic to cover all the details when describing it to a developer. Then, what is the most important part in the architecture?

If we choose Shaw and Garlan's [40] *software architecture* definition, each

---

[3] http://peoria.cs.ualberta.ca/serl

[4] http://peoria.cs.ualberta.ca/aserc

[5] Each team consists of seven people.

"big" part in the picture is a "component", and the lines connecting different components are "connectors". The more connectors a component is associated with, the more interactions it is supposed to have, and the more important role it plays in the system.

The *Persistent Object Manager* (POM) has five different direction-oriented types of connectors, which is the largest among all the components. From this perspective, POM is the most important part in the architecture. And that is why we pay most of our attention to it.

The POM's structure is also a layered system which is shown in Figure 2.4.

POM

Perl DBI

DBD::Avra

Avra RDBMS

Figure 2.4: POM's Structure

The primary interface for database programming in Perl is *database interface* (DBI). DBI is database-independent, which means that it can work with any relational database, such as Oracle, Sybase, Informix, Access, MySQL, etc. The Perl DBI is intended for relational database, and so does not support any OODBMS [12].

DBI separates the actual *database drivers* (DBDs) from the programmer's *application programming interface* (API). Figure 2.5 illustrates the flow of data from a Perl script through to the database.

The data manipulation methods are defined within the DBI module. From there, the DBI module works out which driver should handle the execution of the method and passes the method to the appropriate driver for actual execution. In fact, the DBI module does not perform any database work itself, nor does it even know about any types of databases whatsoever.

This separation of the drivers from the DBI itself makes the DBI a powerful programming interface that can be extended to support almost any relational database available today. The Avra RDBMS is designed and developed by Avra Software Lab Inc[6] and is used in the Prothos framework.

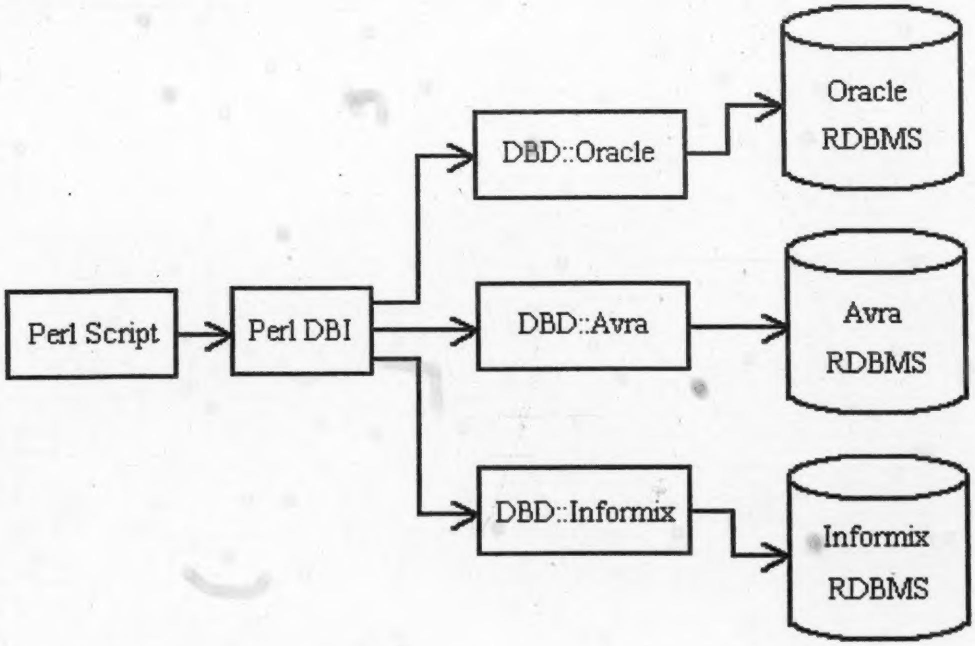
Perl Script
Perl DBI
DBD::Oracle
DBD::Avra
DBD::Informix
Oracle RDBMS
Avra RDBMS
Informix RDBMS

Figure 2.5: Data Flow through DBI

2.3 Formal Methods

Engineering relies heavily on mathematical models and calculation to make judgments about designs. This is in stark contrast to the way in which software systems are designed — with *ad hoc* techniques and ongoing testing. Formal methods bring to software and hardware design the same advantages that other engineering endeavors have exploited: mathematical analysis based on domain models [9].

In a nutshell, formal methods are the applied mathematics of computer systems engineering. They use mathematically based notions to establish models that correspond to the real systems, then to prove models are free of design errors [7]. This verifies the correctness of the abstract model, not the real system though. "Formal" means relating to or involving outward form or structure. Here, it could be further understood that the methods of reasoning are valid by virtue of their form and independent of their content.

---

[6] More information could be found at: http://www.avrasoft.com .

In [4], "formal methods" are interpreted as a generic and inadequate terminology to describe techniques (tools, languages, concepts, ...) aiming at giving more insurance on the behavior of systems before they are built and running.

Formal methods are used to specify and model the behavior of a system and to formally verify that the system design and implementation satisfy functional and safety properties. They express some behavior of the system in a "formally executable" way (the model), using either a software tool to run the model (symbolic testing) or a model checker or theorem prover to certify some properties (model checking).

Formal methods refer to the use of techniques from logic and discrete mathematics in the specification, design, and construction of computer systems (both hardware and software) and rely on a discipline that requires the explicit enumeration of all assumptions and reasoning steps. Each reasoning step must be an instance of a relatively small number of allowed rules of inference. In essence, system verification is reduced to a calculation that can be checked by a machine. In principle, these techniques can produce an error-free design [9].

Although formal methods have a sound basis on mathematics, they are often regarded as impractical [21]. Nevertheless, it is important to realize that formal methods are **not** an all-or-nothing approach. The application of formal methods to the most critical portions of a system is a pragmatic and useful strategy [9].

2.3.1 Model Checking

A *model* is a representation of a set of objects and their relationships. Such a representation is always the result of a process of abstraction.

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not.

Computation Tree Logic (CTL) is defined by Clarke and Emerson [10]. It is a subset of *temporal logic,* which is the generic term for logics that have explicit operators for making statements about changes over time [8].

CTL let us talk about how the state of a system can evolve over time. Usually in our normal operational semantics of machines, we talk about explicit sequences of states, indexed by time. CTL allows us to push the indexing details behind the scenes, but at some loss of expressive power.

CTL formulas consist of the usual atomic propositional logic formulas, plus temporal connectives. The propositional logic formulas are expressions about the state of the system. The temporal connectives are expressions about all possible paths into the future that the state of the system can follow.

Temporal connectives are pairs of symbols. They talk about what can happen from the current state. The "current" state is the one being described in the formula. The future is infinite, that is, the computation does not halt, although it can stay in the same state forever.

Connectives have the following syntax. The first member of the pair is one of:

- A – the universal quantifier, meaning on all paths from the "current" state, read as "inevitably";
- E – the existential quantifier, meaning on at least one path from the "current" state, read as "possibly".

The second member of the pair is one of:

- X – meaning the next state;
- F – meaning some future state;[7]

---

[7] In the CTL temporal logic, the current state is also esteemed a future state.

- G – meaning all future states, read as “globally”;
- U – meaning until.

There are, in total, eight temporal connectives in CTL which are described below. Suppose that the system is in some state S. Consider some CTL formula ϕ: Its truth value depends on the current state S which is determined by the context in which ϕ appears.

1. $\boldsymbol{AX\phi}$ is true if and only if ϕ is true for every immediate successor to state S;
2. $\boldsymbol{AF\phi}$ is true if and only if on all paths into the future from S, there is a state where ϕ holds;
3. $\boldsymbol{AG\phi}$ is true if and only if ϕ is true for every successor to state S, including S, i.e. ϕ is true for all states on all paths into the future from S.
4. $\boldsymbol{A(\phi U\psi)}$ is true if and only if all paths starting in state S satisfy ϕ until reaching a state in which ψ holds;
5. $\boldsymbol{EX\phi}$ is true if and only if ϕ is true for at least one immediate successor to state S;
6. $\boldsymbol{EF\phi}$ is true if and only if there exists a path into the future from S on which there is a state where ϕ holds;
7. $\boldsymbol{EG\phi}$ is true if and only if there is a path from S into the future for which ϕ holds for every state on the path, including S;
8. $\boldsymbol{E(\phi U\psi)}$ is true if and only if there exists a path starting in state S that satisfies ϕ until reaching a state in which ψ holds.

Thus, past time operators are not allowed, and temporal operators cannot be combined directly with the propositional connectives. Note that since all operators are prefixed by A or E, the truth or falsehood of a formula depends only on the given state, and not on the particular branches in the temporal tree that were followed to get to the state. From this perspective, CTL is memoryless.

The *Symbolic Model Verifier* (SMV) system is a tool for checking finite-state systems against specifications in the temporal logic CTL. The input language of SMV is designed to allow the description of finite-state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract [30].

Some pointers are given here, [8] [10] [30], for further reference. They are all related to model checking, temporal logic, CTL, or SMV.

2.4 Related Work

Formal methods are such powerful tools that people in computer science have realized their importance since 1970s. Many projects have applied formal methods to software architecture. For example, IBM's CICS system contains more than a half million lines of code. IBM is using Z to respecify key CICS interfaces to improve its maintainability [33]. These activities are trying to address formal underpinnings for architecture. As new notations are developed, and as the practice of architectural design is better understood, formalisms for reasoning about architectural designs become relevant [40].

Efforts to formalize software architecture began with efforts to apply formal methods directly to individual systems or styles. More recently, there have been some more general treatments of architectural description and analysis. A number of *Architecture Description Languages* (ADLs) and their accompanying tool sets have been proposed as modelling notations to support architecture-based development such as Aesop [16], Darwin [29], Rapide [28], UniCon [39], and Wright [2].

Recently, the work has been done on an architecture interchange language, ACME [17], which is intended to support mapping of architectural specifications from one ADL to another and, hence, enable integration of support tools across ADLs. Although, strictly speaking, ACME is not an ADL, it contains many ADL-like features. Medvidovic and Taylor [31] presented a classification and comparison framework for ADLs to capture highlights in this area.

Before ADLs became popular in the past few decades, research in this field had been developed for a long time. *Module Interconnection Languages* (MILs) and *Interface Definition Languages* (IDLs) were centers of attraction during 1980s. MILs, such as MIL75 [11] and INTERCOL [44], are intended to support the description of large-scale program structure independent of any particular programming language or system. However, MILs are *not* concerned with what the system does (specification information), how the major parts of the system are embedded into the organization (analysis information), or how the individual modules implement their function (detailed design information) [35].

IDLs have been developed to define abstract data interfaces and mappings from programming language constructs into intermediate representations that mediate the transformation between incompatible low-level formats. Examples of these languages include IDL [27], UTS [22], and CORBA's IDL [19]. These languages define a single, restricted model of interaction, typically procedure call or object-method invocation. However, they do not provide a sufficient general basis for the description and analysis of interactions in software architectures.

Although ADLs are definitely a promising direction, it is different from our approach. Instead of defining a descriptive language for the whole software architecture, we are applying formal methods in a domain-specific application framework. The formal models of the architecture in the design phase can be used as a basis for verification of further implementations. We pay our attention to the transaction management component in a specific framework, not the entire architecture. We are intended to present a formal way to understand software architecture in this context at both structural and behavioral aspects.

Another related work is Microsoft's SLAM project [37]. The goal of the project is to check system programs written in C against safety properties using model checking techniques. SLAM introduces a new type of programs called "boolean program" which has only boolean variables in it. The boolean program can take full advantage of *binary decision diagram* (BDD) technique used in model checking. Although boolean program model has proved itself and has been successfully used for checking device drivers, one challenge is how to bridge the gap from a language with pointers (C) to one without pointers (boolean programs). Another problem is the "state space explosion" problem of model checking when it is used in the system programs' context. We do not employ any of the above techniques: boolean program, BDD, etc. We simply build "normal" SMV models based on the current implementation of the framework.

With respect to the software framework, an interesting research is *Framework Constraints Language* (FCL) [24]. Framework constraints are rules that frameworks impose on the code of framework-based applications. FCL helps formally specify these constraints, and further automatically detect errors occurring at the boundary between frameworks and applications. Given the Prothos framework, our formal approach to understanding the transaction management component's behavior would certainly generate a number of crucial framework constraints to help develop Prothos' FCL.

2.5 Thesis Scope

Our work focuses on the transaction management component in the Prothos framework. And among formal methods, we mainly concentrate on model checking technique. By providing a precise semantics for the component at the abstract level, formal models of the system can supply the basis for rigorous, justifiable analysis of critical system properties.

Chapter 3

POM Architecture

We explore the *POM architecture* in this chapter to investigate the goals, structure, and design decisions of this component in the framework. We first focus on the basic data model of POM and how this model is elicited. Then, we build up POM's ontology to provide the consistent terminology throughout the software framework and applications and among the stakeholders. Patterns of using POM are provided and the goal is emphasized. Operations in the basic database level include: query, insertion, deletion, and update. Data integrity constraints provide a means of ensuring that changes made to the database using these basic operations do not lead to a loss of data consistency. A sequence of operations is encapsulated into a transaction.

3.1 Data Models

3.1.1 Object-Oriented Data Model

The object-oriented paradigm is one approach to modelling the data and operations of a computation. The object-oriented approach is ideally suited for many classes of real world problems, especially those having workflow aspects.

There are three fundamental concepts defined in object-oriented paradigm: *object*, *class*, and *inheritance*. An *object* is an entity with a public protocol and a private structure. A *class* is an abstraction of common protocol and a mechanism for object creation. An object is an instance of a class. *Inheritance* captures the commonality hierarchy between classes, and provides shared protocols between objects from different classes in the hierarchy.

By using the object-oriented model, a program consists of a set of messages sent to objects. A problem is solved by sending a message to an object and getting the solution object as a response. Objects have state so they can remember information. Messages are implemented by methods that access state or send messages to other objects [43].

To illustrate different data models, consider a concrete example using *unified modeling language*(UML) [38]. Figure 3.1 depicts the class diagram expressing an association relationship between courses and sections. Each section object is associated with a course. A course can be associated with multiple section objects.

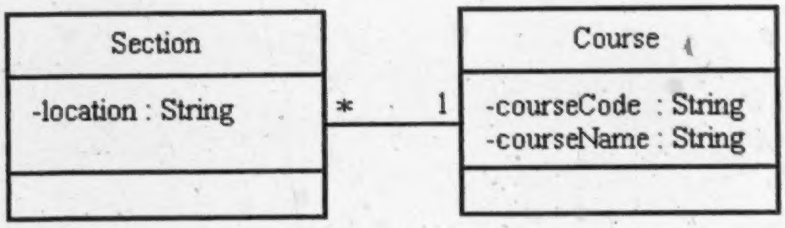

Figure 3.1: A Sample Class Diagram

The class diagram could also be called the object schema. Class diagrams show all the possible relationships. An instance diagram illustrates a specific example of objects satisfying the class relationships.

Figure 3.2 is an instance diagram based on Figure 3.1. An instance diagram uses a real world example, checks the validity of the class diagram, and exposes defects in the model. It shows that the "Algorithms" course (course code is "CMPUT 204") corresponds to three sections whose locations are in "CSC B10", "CAB 377", and "GSB 859" respectively.

3.1.2 Relational Data Model

The mathematical concept underlying the relational model is the set-theoretic *relation*, which is a subset of the Cartesian product of a list of domains. A

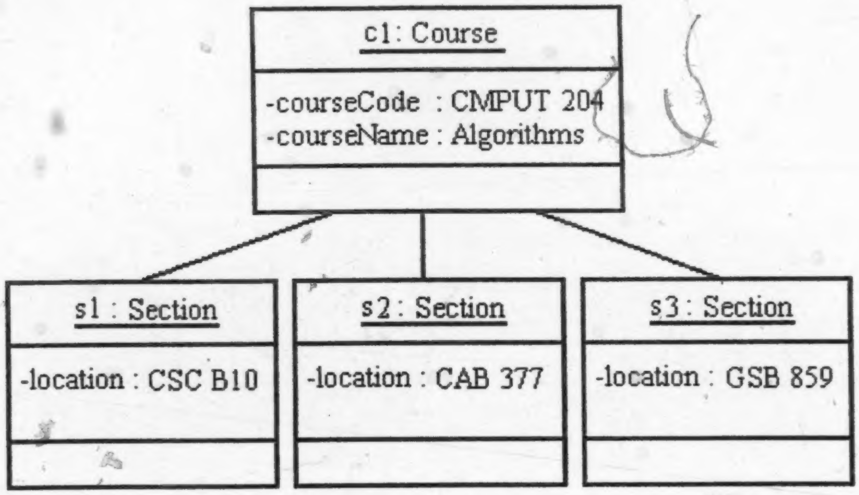

Figure 3.2: A Sample Instance Diagram

relational database consists of a collection of relations. The members of a relation are called *tuples*. It helps to view a relation as a *table*, where each row in a table is a tuple and each column corresponds to an *attribute*.

A relational schema is a description of the structure of relations in a relational database. The schema is a relatively static concept which defines the logical design of the database. Basically, a relation schema is a list of attributes and their corresponding domains. The data values of the relations in the database may change from time to time.

Figure 3.3 shows the UML class diagram of a relational database. There is a "one-to-one correspondence" relationship between the "Table schema" class and "Table" class. A table is created or destroyed by sending messages to a table schema object. Each table contains one or more columns. Rows are manipulated within a table. All four basic database operations, namely query, insertion, deletion, and update, are all defined in the "table" class. A relational data model is nothing but flat tables.

Object-oriented is a programming notion, and an object-oriented model is normally transient and exists only during execution. To allow the model to persist, an *object-oriented database management system* (OODBMS) is re-

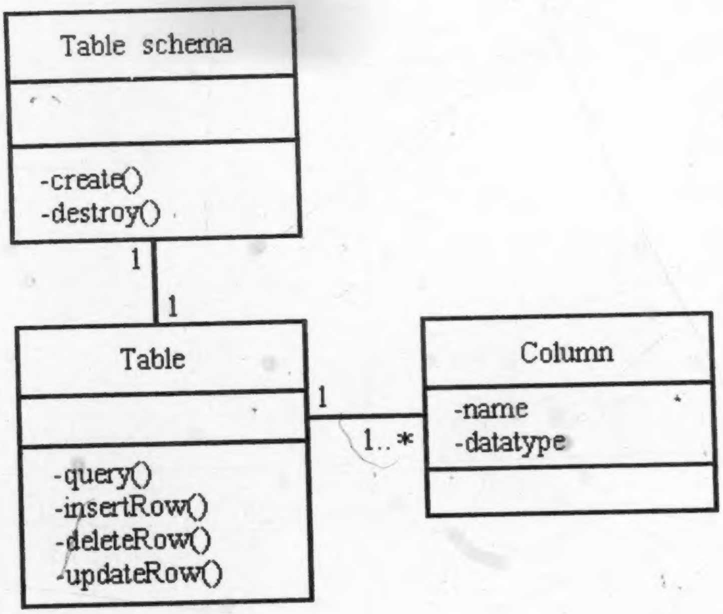
Table schema
-create()
-destroy()
1
1
Table
-query()
-insertRow()
-deleteRow()
-updateRow()
1
1..*
Column
-name
-datatype

Figure 3.3: Class Diagram of Relational Data Model

quired.

An OODBMS integrates database technology with the object-oriented paradigm which was discussed in the prvious section. An OODBMS must consider inheritance and versions of methods. However, there is no real production quality implementation of a pure OODBMS yet.

3.1.3 POM Data Model

POM data model is a *network* model, which can be regarded as an object-oriented model without inheritance and method storage. The associations are transparent in the network model.

A network model consists of a collection of data objects which are connected to one another through links or pointers. Each object is a collection of attributes, each of which contains a data value or an object. A link is an association between precisely two data objects.

The advantage of a network model is that the classes can be easily mapped to relations, and the objects mapped to tuples in the relation.

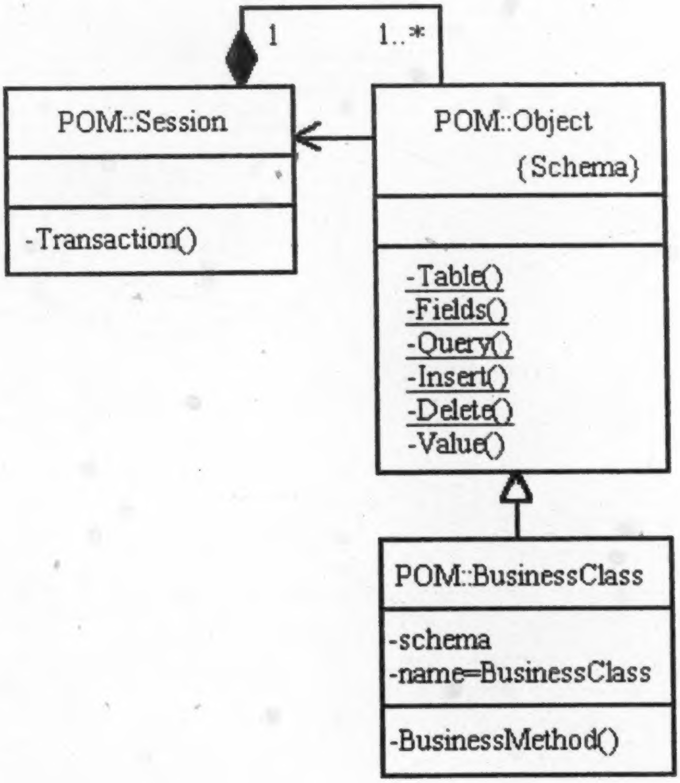
1
1..*
POM::Session
-Transaction()
POM::Object
{Schema}
-Table()
-Fields()
-Query()
-Insert()
-Delete()
-Value()
POM::BusinessClass
-schema
-name=BusinessClass
-BusinessMethod()

Figure 3.4: Class Diagram of POM Data Model

Figure 3.4 shows the class diagram of POM data model. Basically, the POM data model wants to take full advantage of the object-oriented paradigm and database storage mechanism. Thus, OODBMS is an ideal option for POM. However, since there is no real implementation of OODBMS yet, the underlying storage mechanism of POM is adopted from the relational model. So, the goal of POM is to manipulate data objects as in object-oriented programs but they are actually kept around as in relational databases.

3.2 Ontology

In artificial intelligence, an ontology is "the specification of conceptualizations" [20]. It deals with the entities and concepts in the system of interest, as well as the relationships that hold among them. The POM's ontology is the foundation of the approach to understanding the component. It provides the consistent terminology throughout the software framework and applications and among the stakeholders.

Prothos is a database-centric web application framework. It helps developers build their web applications. An application is a collection of business objects and a mechanism for issuing messages to the business objects to change their states.

Table 3.1: Mapping from POM Data Model to Relational Data Model

POM Data Model	Relational Data Model
class	relation
object	row

Table 3.1 provides us the mapping relationship from the POM data model to the conventional relational one. This mapping helps users better understand the terminology used in POM. The "POM::Object" and "POM::BusinessClass" classes are used to define and create business classes and objects. And the "POM::Session" class is used to manipulate the business objects.

The basic idea of POM's object manipulation is to talk to a class to create, locate, and destroy objects. And the modification of the state(s) of an object is defined by talking to the object. All operations must be done inside a transaction in order to guarantee the data consistency.

3.2.1 POM::Object

Conceptual description: POM::Object is the base class of all business classes in POM. It is used to define and create persistent objects that know how to save themselves to persistent storage. This class uses the POM::Session class to interface with the database.

Attribute: The class-scope "Schema". This attribute defines a certain inherent fields of all POM objects, such as: identity(ID), the optimistic locking concurrency control tag (_Delta), creation time (_Created), last update time (_Modified), and so on. The names in parentheses are fields' names appeared in POM::Object.

Methods:

- Table(): A class method to get or set the name of database table for a persistent class. The specified parameter is used to name the database table.
- Fields(): A class method to get or set the field definitions for a persistent class. When defining a database table, the parameter of this method acts as a table schema in the relational database management system.
- Query(): A class method to return a set of persistent objects in an array satisfying the selection condition(s). The basic format of query conditions in POM is: *where P sort by S*. The *where* clause is comprised of a predicate involving fields of the persistent class. And the *sort by* clause causes the objects in the result array of this query to appear in sorted order.
- Insert(): A class method to create a new object for the given class to be inserted into the database.
- Delete(): A class method to delete an existing object from the given class in the database. Note that this method checks referential integrity before deleting the object, that is, an exception will occur if the object being deleted is referenced by other object(s). We will be further investigating data integrity issues in Section 3.4.
- Value(): An instance method to return or update the specified field of an object. Compared to the relational data model shown in Figure 3.3, the update method in POM is an instance method, which means a particular

data object has to be identified before its field(s) can possibly be changed. In the relational model, the update method is sent to table directly, which is a class method.

Notice that all basic database operations are defined within a POM::Session transaction method. In other words, Query(), Insert(), Delete(), and Value() methods must be called within a transaction in order to guard the data integrity and system constraints.

3.2.2 POM::BusinessClass

Conceptual description: POM::BusinessClass inherits from POM::Object class to define and create persistent business classes.

Attributes:

- schema: The business class description. The schema defines fields' names and types of the business class. Since the class-scope "Schema" attribute defines inherent fields for all POM objects, the "schema" attribute here is an extension of "Schema". Notice that in both "Schema" and "schema" attributes, each field must have the *name* and *type* information.

- name: The name of the business class, or from the relational model's standpoint, the table name or relation name.

Method: BusinessMethod(). This allows users to define business method(s) corresponding to this particular business class.

3.2.3 POM::Session

Conceptual description: POM::Session class provides the manipulation mechanism for POM objects. A transaction is a sequence of basic database operations that performs a single logical function. Operations defined within a transaction are inseparable. A transaction must guarantee the *atomicity* requirement, that is, all operations inside a transaction happen successfully or nothing happens.

Method: Transaction(). The method is used to define and execute a transaction in order to manipulate POM objects. The sequence of basic database operations of the transaction are defined in the method body.

3.2.4 Relationships

In Figure 3.4, there are three relationships among classes: inheritance(the triangle arrow), composition(the dark diamond arrow), and navigation(the arrow).

The inheritance relationship means that every business class is a POM object. Composition indicates that each POM::Session object has one or many POM objects. These objects are involved in this session's current transaction. Navigation represents that a POM object knows which POM session it belongs to.

3.3 Patterns of Use

A software framework imposes many patterns when using it to development applications. Given the discussion of POM ontology in Section 3.2, we generalize some patterns of using POM and provide a code-based example in order for users to better understand these patterns.

1. Users must define a POM class before using it. When defining a POM class, users must provide the name and type information for each field of the class. This is embodied in the "schema" attribute.

2. The "name" attribute of business classes is unique. Different business classes must have different names.

3. In the "schema" attribute of business classes, name of fields is unique. Besides, no field can have the same name as the ones defined in "Schema" attribute of POM::Object class.

4. All basic database operations, namely query, insertion, deletion, and update, must be called within a transaction procedure.

5. The query, insertion, and deletion messages are sent to classes while the update message is sent to objects.

Based on the course-section example introduced in Section 3.1, we give a detailed code-based example to help understand the patterns of use. The following code fragment defines the *Class::Course*[1] class:

⋮

```
# Class definition
package Class::Course;
use POM::Object;
@Class::Course::ISA = “POM::Object”;
Class::Course → Table(“course”);
Class::Course → Fields([CourseCode ⇒ “S”[2]],
                       [CourseName ⇒ “S”]);
```

⋮

The following code fragment defines the *Class::Section* class, given the *Class::Course* class has already been defined in the system:

⋮

```
# Class definition
# The Class::Course has already been defined
package Class::Section;
use POM::Object;
@Class::Section::ISA = “POM::Object”;
Class::Section → Table(“section”);
Class::Section → Fields([Course ⇒
                            “Class::Course”],
                        [Location ⇒ “S”]);
```

---

[1] “Class::” is the conventional prefix for all business classes in the Prothos framework [42].

[2] “S” is the abbreviation of type “String” defined in the Prothos framework [42].

⋮

Suppose we have inserted several data objects to *Class::Course* and *Class::Section* classes already. Note that all insertion operations must be called within a transaction. Due to renovations to the GSB building, all sections scheduled in "GSB 859" are now moved to "CSC B02". The following code fragment illustrates how to perform such a task inside a transaction:

⋮

```
# Start of a transaction
# Select all sections located in "GSB 859"
my @sections = Class::Section → Query(
                        [Where ⇒ "Location='GSB 859' "],
                        [SortBy ⇒ "Course"]);
# Update all selected sections with new location
foreach my $section (@sections) {
    $section → Value("location", "CSC B02");
}
# End of a transaction
```

⋮

To conclude this section, we emphasize the goal of POM again. POM is to manipulate data objects as in object-oriented programs but they are actually kept around as in relational databases.

3.4 Data Integrity Issues

Integrity constraints ensure that changes made to the database do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates are too costly to test. Thus, we usually limit ourselves to integrity constraints that can be tested with minimal overhead.

3.4.1 Domain Constraints

We have argued earlier in Section 3.2 that users must provide the name and type information for each field of the class. In other words, a domain of possible values must be associated with each attribute. When a POM class is defined in Prothos, the type information is tightly connected with the field name.

Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data object is entered into the database, or the user wants to change the value of some existing data objects.

In Prothos' context, domain constraints would further assist us with restricting the use of data objects correctly. For instance, it would be meaningless to send an "arithmetic multiplication" message to a "String" typed data field.

3.4.2 Referential Integrity

In the POM model, referential integrity means that all pointers or links resolve. In another word, an data object cannot be deleted if some other objects point to it.

POM has offered its own way to declare the referential integrity as explained in Section 3.2 and Section 3.3. Database modifications can cause violations of referential integrity. POM guards the constraint by disallowing the data object to be deleted if there exists any data object that has a link to it. In another word, as long as associations are maintained by links, referential integrity is assured.

3.4.3 Functional Dependencies

Functional dependencies are a generalization of key dependencies. They require that the value for a certain set of fields determines uniquely the value for another set of fields.

Let $\alpha \subseteq schema\ of\ a\ business\ class$ and $\beta \subseteq schema\ of\ a\ business\ class$. The *functional dependency*:

$$\alpha \rightarrow \beta$$

holds on the business class if for all pairs of objects o_1 and o_2 in the business class such that $(o_1 \rightarrow Value(\alpha) = o_2 \rightarrow Value(\alpha))$, it is also the case that $(o_1 \rightarrow Value(\beta) = o_2 \rightarrow Value(\beta))$.

In order to understand the concept of functional dependencies, consider the schema:

$$Club\text{-}schema = (club\text{-}name,\ city,\ province).$$

Since each city is associated with precisely one province[3], the following functional dependency holds:

$$city \rightarrow province$$

Whenever the city of "Edmonton" appears as the *city* field's value in an object, the value of the *province* field must be "Alberta" of the same object.

Thus, functional dependencies allow us to express facts about the enterprise that we are modelling with the database. In Prothos, it is the responsibility of **business class methods** to protect this kind of integrity constraints.

---

[3] In this example, no two cities have the same name.

Chapter 4

Transaction Model in Prothos

A *transaction* is a sequence of operations that performs a single logical function in a database application. The basic database operations are query, insertion, deletion, and update introduced in Chapter 3. For a single transaction, the most prominent property is **atomicity**.

The *atomicity* is also known as the "all or none" requirement. The basic operations are gathered together to form a transaction because they carry out one single task and are inseparable. An example is funds transfer, in which one account is debited and another is credited. Clearly, it is vital that either both the credit and debit occur, or that neither occurs. That is, the funds transfer must happen in its entirety or not at all.

This chapter deals with non-concurrency issues. The concurrent transactions will be discussed later in Chapter 5.

4.1 Data Object Message Simplification

The basic database operations presented in Chapter 3 include query, insertion, deletion, and update. However, since transactions are programs, it is computationally difficult to determine the exact operations that a transaction performs and how operations of various transactions interact.

For this reason, the basic operations could be simplified to **read** and **write**. This simplification makes sense because ultimately each transaction can be viewed as a read/write sequence.

In the Prothos transaction model, we adopt the notations "**read**(O_i)" and "**write**(O_i)" to represent the "read" and "write" messages sent to an arbitrary data object O_i respectively.

4.2 The Storage Hierarchy

The Prothos storage hierarchy is shown in Figure 4.1.

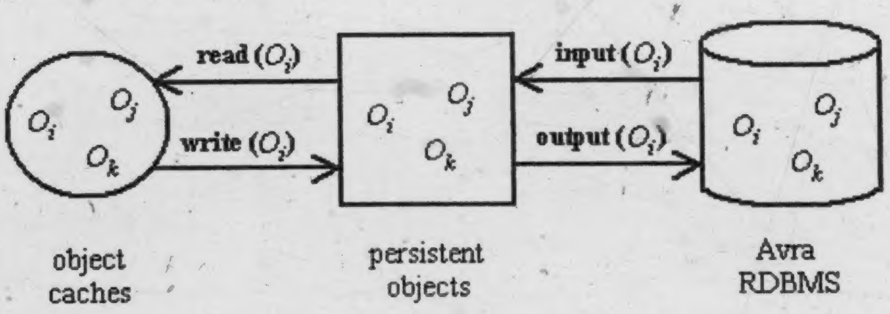

Figure 4.1: The Storage Hierarchy

The objects in object caches are actual objects in memory manipulated by the object-oriented programs. They form the application, which is a collection of business objects and a mechanism for issuing messages to the business objects to change their states.

The program talks to POM to obtain data objects, and further send them back. POM is depicted in Figure 4.1 labeled as persistent objects. The Avra RDBMS is the bottom layer used to store the data objects in the database.

Two pairs of operations transfer data objects between different storage layers.

- **input**(O_i), which transfers the physical block in which data object O_i resides to the persistent objects storage.
- **output**(O_i), which writes O_i back to the RDBMS layer.
- **read**(O_i), which makes a copy of the data object O_i to the local work space in order for the transaction program to handle.

- **write**(O_i), which assigns the local copy of O_i back to the persistent objects layer.

Notice that Prothos does a mapping between the Avra RDBMS and the object-oriented persistent layer as indicated in Section 2.2.3.

4.3 The Transaction Model

A transaction is a program unit that accesses and possibly updates various data objects. Each transaction is a unit of atomicity. In Prothos, transactions must not violate any database consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates.

4.3.1 The State Diagram

The finite-state diagram corresponding to a transaction is shown in Figure 4.2.

It is fairly straightforward to recognize that the transaction model illustrated in Figure 4.2 is a *nondeterministic finite automaton* (NFA). The formal definition chosen from [18] is:

$$M = (Q,\ \Sigma,\ \Delta,\ q_0,\ F)$$

where M stands for "machine", i.e., the transaction model in our case. The five-tuple components are specified as follows:

1. Q— a finite, nonempty set of states. In this model, the states

$$Q = \{active,\ partially\ committed,\ failed,\ committed,\ aborted\}$$

The following descriptions render the semantics of each state:

- **Active**, the initial state. A transaction is in the active state when it has started execution.
- **Partially committed**, after the last statement has been executed.

- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
- **Committed**, after "successful" completion.

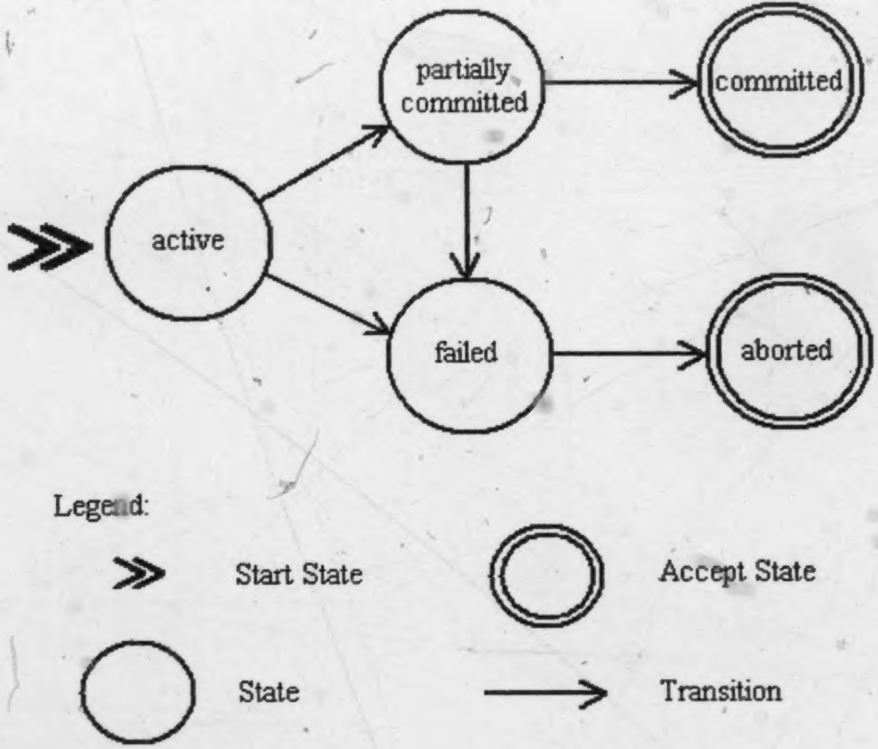
partially committed
committed
active
failed
aborted
Legend:
Start State
Accept State
State
Transition

Figure 4.2: The Transaction State Diagram

2. Σ— the data alphabet. There is no input character defined in our machine explicitly. Transactions occur between states as a result of time passing and events occurring during execution.

3. Δ— the transition relation, which is a finite subset of

$$Q \times \Sigma^* \times Q$$

Table 4.1 designates Δ for the above NFA. A transition table is a convenient method for representing the transition relation of an NFA. The transitions are numbered for convenience, but this numbering is not part of the transition relation.

Table 4.1: The Transaction Model's Transition Table

Transition Number	**State**	**New State**
1	active	partially committed
2	active	failed
3	partially committed	failed
4	partially committed	committed
5	failed	aborted

Note that we ignore the **"input symbol"** in Table 4.1 since the data alphabet, Σ, is implicitly defined over the clock tick. The time unit of any transition is greater or equal to one, indicating arbitrary input (any non-zero time interval) transitions are incorporated in this model.

Although Λ-transitions are allowed in NFAs, they are prohibited by our machine. A transaction is in and only in one state at a time, and cannot enter to another state without consuming a certain period of time.

4. q_0— the initial state or start state, $q_0 \in Q$. As we can see in Figure 4.2, the initial state is *active*, i.e., $q_0 = active$.

5. F— the set of accepting states, $F \subseteq Q$. $F = \{committed,\ aborted\}$ in our model. These two states do not have any "next state". This implies that once a transaction enters into either state, it terminates immediately, whether the transaction is successfully committed, or totally rolled back. This actually assures the "atomicity" property of a transaction.

Based on this model, every transaction starts in the *"active"* state. A transaction moves into the *"partially committed"* state when it reaches its last statement. At this point, all the operations have been done for this transaction, but it is still possible that the transaction may have to be aborted since the actual output may not have been written to the persistent storage yet.

A transaction goes into the *"failed"* state after it is determined that the normal execution cannot proceed any longer. Such a transaction must be rolled back, and then finishes up with the *"aborted"* state. In Prothos, an error exception is generated at this point by the framework, and it is completely the choice of the user to restart the transaction or simply kill it.

Finally, if the transaction ends up with either *"committed"* or *"aborted"* state, it terminates properly. Such a *trace* is accepted by our NFA machine, and the transaction itself does not interfere with any database constraints in that among the whole state space, only these two states preserve the system consistency. When the machine stops in a nonaccepting state, it signifies that such a transaction is rejected. In another word, the Prothos' transaction model does not permit this kind of transactions to happen as they may destroy the system consistency constraints.

4.3.2 Model Checking The Transaction Model

Model checking is an appropriate method to reason the behavior of the transaction model. We employ the *computation tree logic* (CTL), and build up a *symbolic model verifier* (SMV) model in order to observe crucial specifications of Prothos' non-concurrent transaction model. Figure 4.3 illustrates the fundamental ideas of model construction.

The important and interesting specifications of the SMV model are listed as follows:

$$AG(state = failed \rightarrow AX(state = aborted)) \tag{4.1}$$

Spec 4.1 reveals that for all nodes in the tree, the next state of "failed" must be "aborted".

$$AG(AF(state = committed \mid state = aborted)) \tag{4.2}$$

Spec 4.2 states that every branch has either "committed" or "aborted" as its ending state.

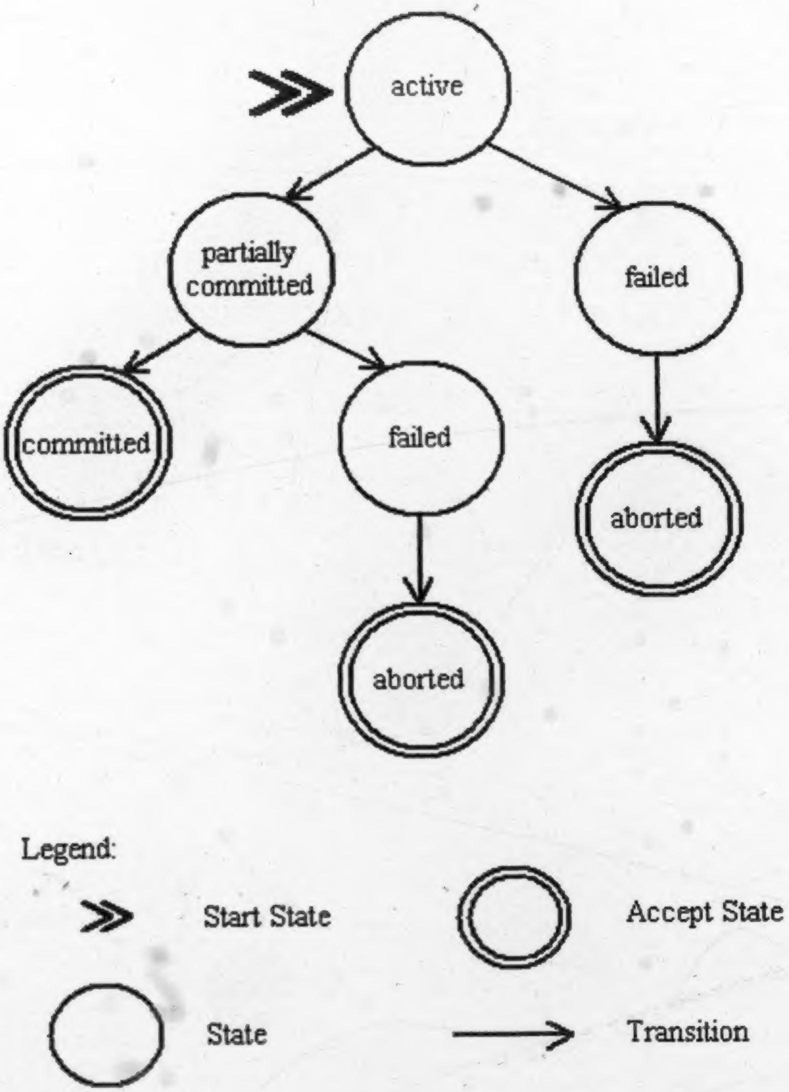
active
partially committed
failed
committed
failed
aborted
aborted
Legend:
Start State
Accept State
State
Transition

Figure 4.3: The CTL Diagram of The Transaction Model

$$AG(state = failed \rightarrow AG!(state = committed)) \tag{4.3}$$

$$AG(state = committed \rightarrow AG!(state = failed)) \tag{4.4}$$

The conjunction of **Spec 4.3** and **Spec 4.4** proclaims that "failed" and "committed" states cannot appear in one computational branch.

$$AG(state = aborted \rightarrow AG!(state = committed)) \tag{4.5}$$

$$AG(state = committed \rightarrow AG!(state = aborted)) \tag{4.6}$$

And the conjunction of **Spec 4.5** and **Spec 4.6** indicates that "aborted" and "committed" states contradict each other in any branch.

Although the above specifications provide us a formal description of the properties in the Prothos' transaction model, it is easier to extract same possessions from the model straightly using natural languages. The model has only seven states and six transitions, which is too simple to bother the model checking technique.

In consequence, model checking does not fully come into play in the context of the non-concurrent single transaction model. In other words, the basic transaction model is designed to be obviously correct to developers.

Chapter 5

Concurrency Control

It is undoubtedly necessary for Prothos to provide the concurrency control among transactions that are executed simultaneously since a web application on top of Prothos could ordinarily have many clients access the service at the same time.

The benefits of a concurrent-transaction-supportive environment are increased processor utilization and higher total transaction *throughput* — that is, the amount of work which is accomplished in a given period of time.

We have already considered Prothos' basic transaction model's atomicity in the absence of concurrency in Chapter 4. Nevertheless, when several transactions execute concurrently, the consistency of the database may no longer be preserved. In this chapter, we pay our attention to how Prothos controls the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the system.

5.1 Schedules

When a number of transactions run concurrently, database consistency can be violated despite the correctness of each individual transaction. In this section, we review the concept of *serializability* to help identify those executions that guard the system consistency.

Consider a simple banking scenario: there are two accounts A and B with the original balances of \$100 and \$200 respectively. Let Tr_1 and Tr_2 be two transactions that transfer funds from one account to another. Transaction Tr_1

transfers \$20 from account A to account B and is defined in Table 5.1.

Table 5.1: Transaction Tr_1

```
Tr1 :  read(A)
       A = A - 20
       write(A)
       read(B)
       B = B + 20
       write(B)
```

Transaction Tr_2 transfers \$40 from account B to account A and is defined in Table 5.2.

Table 5.2: Transaction Tr_2

```
Tr2 :  read(B)
       B = B - 40
       write(B)
       read(A)
       A = A + 40
       write(A)
```

A *schedule* is an execution sequence that stands for the chronological order in which basic operations are performed in the system. Suppose that these two transactions are executed concurrently in the order Tr_1 followed by Tr_2. Table 5.3 exhibits such a schedule with chronological order from top to bottom. We use the convention that operations of Tr_1 appear in the left column and those of Tr_2 are to the right.

After the execution in Table 5.3, the eventual values of accounts A and B become \$120 and \$180 respectively. In this example, the total amount of money in A and B should remain unchanged after the execution of both transactions. That is to say the preserved sum, $A + B = \$300$, is the consistent constraint in this case.

A schedule for a set of transactions must consist of all operations of those transactions, and must maintain the order in which the operations appear in

Table 5.3: A Serial Schedule (in which Tr_1 is followed by Tr_2)

Tr_1	Tr_2
read(A)	
A = A - 20	
write(A)	
read(B)	
B = B + 20	
write(B)	
	read(B)
	B = B - 40
	write(B)
	read(A)
	A = A + 40
	write(A)

each individual transaction.

The schedule displayed in Table 5.3 is a *serial schedule*. Each *serial schedule* consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule [25]. Thus, for a set of n transactions, there are $n!$ valid serial schedules in total.

Table 5.4: A Serial Schedule (in which Tr_2 is followed by Tr_1)

Tr_1	Tr_2
	read(B)
	B = B - 40
	write(B)
	read(A)
	A = A + 40
	write(A)
read(A)	
A = A - 20	
write(A)	
read(B)	
B = B + 20	
write(B)	

Table 5.3 delineates one of the two serial schedules in our sample. The other one is shown in Table 5.4. Again, the sum $A + B$ is preserved as expected.

When several transactions are executed concurrently, operations from various transactions could be interleaved. In this case, the schedule is not serial any more. Thus, the number of possible schedules for a set of n transactions is much larger than $n!$. The schedule in Table 5.5 is not a serial schedule. However, after the execution takes place, the sum $A + B$ is indeed preserved.

Table 5.5: A Serializable Schedule

Tr_1	Tr_2
	read(B)
	B = B - 40
	write(B)
read(A)	
A = A - 20	
write(A)	
	read(A)
	A = A + 40
	write(A)
read(B)	
B = B + 20	
write(B)	

Not all concurrent schedules result in a consistent state. To demonstrate, consider the nonserial schedule in Table 5.6. After the execution of this schedule, the balances of accounts A and B become \$80 and \$180 respectively. Clearly, this is an inconsistent state since the sum of $A + B$ is not equal to \$300 any more.

In this schedule, the "write" operation of account A in transaction Tr_1 has been "delayed". During that time, Tr_2 transfers \$40 from account B to A, leading the temporary balances of A and B to be \$140 and \$160 respectively. At this point, the system is consistent. Then, when Tr_1 resumes to execute, it overwrites the value of account A, making it to be \$80. Tr_1 also obtains the new balance of B, and adds \$20 to it. So this schedule ends up with an inconsistent state with the final values of accounts A and B being \$80 and \$180 respectively.

Table 5.6: A Nonserializable Schedule

Tr_1	Tr_2
read(A)	
A = A - 20	
	read(B)
	B = B - 40
	write(B)
	read(A)
	A = A + 40
	write(A)
write(A)	
read(B)	
B = B + 20	
write(B)	

All serial schedules apparently hold the consistent constraints since the atomicity of each participant transaction is guaranteed. Needless to say, a schedule, after execution, must leave the database in a consistent state. In another word, the schedule should, in some sense, be equivalent to a serial schedule.

We say that a schedule is ***serializable*** if it produces the same outcome as a serial schedule. Schedules presented so far are all serializable except for the one in Table 5.6. Only by offering protection of serializability could a concurrent control protocol be correct.

5.2 Lock-Based Protocols

One way to ensure serializability is to require that access to data objects be done in a ***mutually exclusive manner***; that is, while one transaction accesses a data object, no other transaction can modify that object. The principal way to implement this is to allow a transaction to access a data object only if it is currently holding a **lock** on that object.

5.2.1 Locks

There are two modes of locks to a data object:

- **Shared**, also known as **optimistic**. If a transaction Tr_i has obtained a shared-mode lock (denoted by S) on object O, then Tr_i can read this object but it cannot write O.
- **Exclusive**, also known as **pessimistic**. If a transaction Tr_i has obtained an exclusive-mode lock (denoted by X) on object O, then Tr_i can both read and write O.

Given the above descriptions, Table 5.7 illustrates the relationship between these two modes of locks ($Tr_i \neq Tr_j$).

Table 5.7: The Relationship Between Locks

Tr_i requests	Tr_j holds	Tr_i acquires immediately
S	S	Yes
S	X	No
X	S	No
X	X	No

In order to access a data object, a transaction Tr_i must lock it first. If the object has not been locked yet, or locked by the same transaction, Tr_i could obtain the requesting lock mode immediately. If the object has already been locked by another transaction Tr_j, from Table 5.7, only in one situation could Tr_i acquire the desired lock instantly, i.e., the requesting mode is S, and the currently held mode by Tr_j is S. Under other circumstances, Tr_i must wait. A transaction may release a lock on a data object which it had locked at some earlier point. So if a transaction obtains a share-mode lock on a data object, others can read but cannot write that data object. If a transaction issues an exclusive lock on a data object, others cannot read or write it.

We introduce the notations "lock-S(O)", "release-S(O)", "lock-X(O)", and "release-X(O)" to represent requesting a shared lock, releasing a shared lock, requesting an exclusive lock, and releasing an exclusive lock on the data object O respectively.

The example elaborated in Section 5.1 has to be extended with these new instructions so that the locking scheme would be mirrored in schedules. If we do not set up any restrictions on locks, the concurrent execution schedule in Table 5.8 is possible.

Table 5.8: A General Locking Schedule (which causes inconsistency)

Tr_1	Tr_2
lock-S(A)	
read(A)	
release-S(A)	
A = A - 20	
	lock-S(B)
	read(B)
	release-S(B)
	B = B - 40
	lock-X(B)
	write(B)
	release-X(B)
	lock-S(A)
	read(A)
	release-S(A)
	A = A + 40
	lock-X(A)
	write(A)
	release-X(A)
lock-X(A)	
write(A)	
release-X(A)	
lock-S(B)	
read(B)	
release-S(B)	
B = B + 20	
lock-X(B)	
write(B)	
release-X(B)	

In the above schedule, operations except for those locking instructions are in the same order as the schedule in Table 5.6. Thus, such a schedule brings the system to an inconsistent state with the values of A and B being \$80 and \$180 in the end, which violates serializability.

One potential reason for this is that both transactions should have only issued exclusive locks on data objects A and B since they change the values and eventually write them back. Another error-prone factor for the schedule is that one transaction releases the lock so early that another transaction may face an inconsistent state.

Suppose now that both transactions issue exclusive locks solely and releasing is delayed to the end of each transaction. The following schedule generates a new problem.

Table 5.9: A General Locking Schedule (which causes deadlock)

	Tr_1	Tr_2	
▷	lock-X(A)		
	read(A)		
	A = A - 20		
	write(A)		
		lock-X(B)	◁
		read(B)	
		B = B - 40	
		write(B)	
		lock-X(A)	waiting because of ▷
		read(A)	
		A = A + 40	
		write(A)	
		release-X(B)	◇
		release-X(A)	
waiting because of ◁	lock-X(B)		
	read(B)		
	B = B + 20		
	write(B)		
◇	*release-X(A)*		
	release-X(B)		

Since Tr_1 is holding an exclusive lock on A (marked ▷ in Table 5.9) and Tr_2 is requesting an exclusive-mode lock on A, Tr_2 is waiting for Tr_1 to release A. Likewise, since Tr_2 is locking account B in an exclusive mode (marked ◁) and

Tr_1 is requesting B, Tr_1 is waiting for Tr_2 to release B. As a consequence, we have arrived at a state where neither these transactions can ever proceed with its normal execution. This situation is called *deadlock*.

Deadlock is a state in which processes are blocked waiting for something that will never happen. Consider two processes, $Proc_1$ and $Proc_2$, that require resources A and B. If they attempt to acquire these resources in the opposite order, as shown in Table 5.10, then there is the potential for deadlock:

Table 5.10: A Sample Deadlock Situation

$Proc_1$:	get A	$Proc_2$:	get B
	get B		get A

If $Proc_1$ succeeds in acquiring resource A and $Proc_2$ succeeds in acquiring resource B, then they are deadlocked. In general there are four conditions that must be present before deadlock can occur:

1. Processes need to share resources that must be used under mutual exclusion.
2. Processes hold on to resources already allocated to them while waiting for more.
3. Resources are non-preemptible.
4. A circular chain of requests and allocations exists.

We can prevent deadlock by removing any of the four conditions from the system:

1. **Condition 1:** Don't allow process access to resources that must be used under mutual exclusion. A common example of this approach is having a spool directory for printing and putting the printing of documents into the heads of a printer daemon rather than letting processes print directly.
2. **Condition 2:** Make processes request all resources they require at once.

3. **Condition 3:** Add a mechanism for removing a resource from a process that has it. This can waste work though, because a process that is nearly finished may be required to relinquish a resource and start again later.

4. **Condition 4:** Order the resources and force processes to acquire the resources they require in order, making it impossible for a circular chain of requests and allocations to exist.

If deadlock cannot be avoided, a deadlock detection and recovery mechanism must be employed to find the deadlock and further recover from it.

Given the above discussion about deadlock, we can see that the *italic operations* in Table 5.9 can never get running because of the deadlock status. When deadlock occurs, one of the transactions must be rolled back so that the data objects that were once locked by that transaction are released. These data objects are then available to other transactions, which can continue with their executions.

Deadlock freedom is indispensable and critical for a concurrency control protocol since it is very dangerous for a system to enter a deadlock state. Even though a protocol does not ensure freedom from deadlock, it must supply a roll back mechanism if deadlock happens.

As we could learn from this section, the general locking protocol, which has no restrictions on locks, can guarantee neither serializability nor deadlock freedom. Naturally, we would impose a set of rules on how a transaction may lock and release data objects. We hope that the protocol ensures both serializability and freedom from deadlock when applying locks in a cautious manner.

5.2.2 The Two-Phase Locking Protocol

One popular locking protocol that ensures serializability is the *two-phase locking protocol*. It was first introduced by Eswaran, et al. [13]. This protocol requires that each transaction issue lock and release requests in two phases:

- **Growing phase**. A transaction may obtain locks but may not release any lock.

- **Shrinking phase**. A transaction may release locks but may not obtain any new locks.

A transaction starts within the growing phase initially. The transaction acquires locks as needed. Once the transaction begins to release a lock, it enters the shrinking phase and no more lock requests may be issued.

The two-phase locking protocol ensures serializability. Some formal proofs could be found in [25], [41]. However, it does not guarantee freedom from deadlock. To demonstrate this, observe that the schedule in Table 5.9 is a perfectly legal two-phase locking schedule. Both transactions get into the shrinking phase at the ◇ sign. As argued in Section 5.2.1, this schedule leads the system to a deadlock state.

As a rule, the two-phase locking protocol allows a transaction to lock a new data object only if it has not yet released any data object. In the absence of information concerning the manner in which data objects are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability [25]. Roll back mechanism needs to be appended when applying the two-phase locking protocol because the protocol does not ensure deadlock freedom.

5.2.3 Prothos' Optimistic Concurrency Control Mechanism

Before introducing Prothos' concurrency control scheme, let's consider Prothos' concurrency architecture first.

Figure 5.1 shows an example of three concurrent processes in Prothos. Each application corresponds to one POM object, and each POM object defines a transaction. All three processes connect to Avra RDBMS at the bottom.

There is not just one POM instance running as a central control of concurrent transactions. Instead, there may be many POM objects running at the same time in the system. So, the RDBMS layer becomes the only possible place to coordinate concurrent transactions. And that is why Prothos adopts the locking scheme to control concurrency.

Prothos uses an optimistic mechanism as its concurrency control scheme, which is an enhanced version of the two-phase locking protocol. The basic idea is that transactions issue optimistic locks most of the time. A transaction merely issues an exclusive-mode lock on a data object it needs to write back to the persistent storage when the transaction in the end enters the "commit" phase. The exclusive-lock requests indicate the start of "commit" phase.

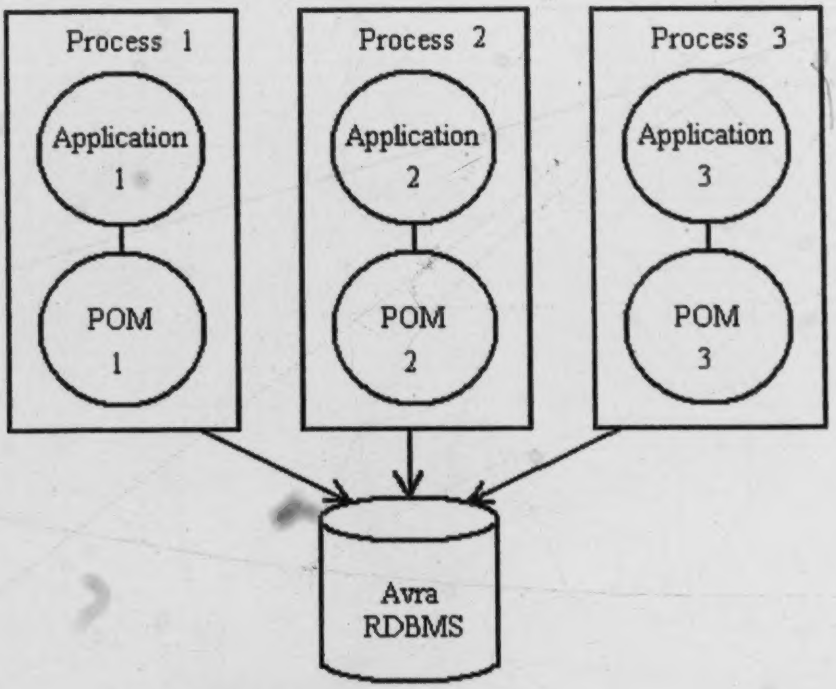

Figure 5.1: Concurrency Control in Prothos

Every transaction abides by the "growing" and "shrinking" rules defined in the two-phase locking protocol. However, the semantics of lock modes introduced in Section 5.2.1 have been modified.

If a transaction Tr_i has obtained a shared-mode lock on object O, then Tr_i can read and write this object. At the same time, other transactions may read O and possibly write O. The transaction must issue the exclusive lock on O first in order to write O. If the transaction has held a shared lock on O already, it will *upgrade* the shared-mode lock to exclusive-mode lock. And when the exclusive lock is released, the shared lock is also gone. If a transaction Tr_i has obtained an exclusive-mode lock, then Tr_i can both read and write O. And no other transaction can read or write it. The point here is that for any data object, if there is an exclusive lock on it, there can only be one transaction holding that lock at a time in the system.

Given the above descriptions of POM's locks, Table 5.11 illustrates the relationship between these two modes of locks (T[illegible]$\neq Tr_j$).

Table 5.11: The Relationship Between Locks in POM

Tr_i requests	Tr_j holds	Tr_i acquires immediately
S	S	Yes
S	X	No
X	S	**Yes**
X	X	No

Compared to Table 5.7, the only difference here is emphasized by the boldface **Yes**. That is the requesting transaction can obtain the exclusive lock immediately even though the data object is currently shared-locked by another transaction. This makes the Prothos' concurrency control mechanism less strict. Intuitively, it leads to more throughput than the two-phase locking protocol does. We will show an example later on to demonstrate this assertion is true.

Notice that we use all lock-related notations henceforth to represent the locks defined in Prothos, not the ones in Section 5.2.1.

Prothos implements its optimistic concurrency control scheme in the following way:

- Every data object has an intrinsic attribute called "_Delta" as we discussed in Chapter 3. The *persistent object manager* (POM) defines this attribute for every database schema in Prothos. "_Delta" has the type of integer, and is hidden from the user. The value of "_Delta" is always increasing for the purpose of identifying changes to the data object. For the data object O, we use Δ_O to denote the value of "_Delta" in O.

- If a transaction wants to read a data object during its execution, it requests a shared-mode lock. If the object is at present locked exclusively by another transaction, the requesting transaction has to wait. A timeout is set up for the waiting transaction to avoid an arbitrarily long inactivity. When the timeout expires, the requesting transaction has to abort.

- When a transaction successfully issues a shared-mode lock on an object, it is able to read the object. In Prothos, the transaction reads not only the values of actual attributes, but also the value of the attribute "_Delta" in the object. These two reads must happen atomically. For each transaction, Prothos locally keeps the most recent value of "_Delta" of each data object the transaction accesses.

- After a write-transaction finishes performing all its basic operations, it needs to write all involved data objects back. The transaction requests the exclusive-mode lock to every participant object. If any of the objects is currently holding an exclusive lock issued by another transaction, the requesting transaction has no choice but to wait. Similarly, a timeout is established in order to prevent the requesting transaction from waiting forever.

- When a transaction successfully issues the exclusive-mode locks on all its participant objects, it enters the "commit" phase. Before it actually writes back all objects, it compares its local copy of each object's "_Delta" value to the one retrieved from the database. If all local copies of "_Delta" are equal to the latest ones from the database, it means no other transactions have ever changed these data objects, and the transaction can safely write the objects back. At the same time, each object's "_Delta" value is increased to mark this transaction's completion. Otherwise, it aborts since some objects have been modified during its execution.

- While a transaction holds an exclusive-mode lock on a data object, no other transaction can read or write it. All other locking requests have to wait till the exclusive lock is released. If a transaction aborts or successfully commits, it releases locks on all invloving data objects.

To conform to the optimistic locking protocol in Prothos, the two transactions developed in the example in Section 5.1, Tr_1 and Tr_2, are now adjusted to what is shown in Table 5.12 and Table 5.13 respectively.

Table 5.12: Transaction Tr_1 in Prothos

Tr_1 :	lock-S(A)
	read(A and Δ_A)
	A = A - 20
	lock-S(B)
	read(B and Δ_B)
	B = B + 20
	lock-X(A and B)
	compare(Δ_A and Δ_B)
	write(A and Δ_A)
	write(B and Δ_B)
	release-X(A and B)

Table 5.13: Transaction Tr_2 in Prothos

Tr_2 :	lock-S(B)
	read(B and Δ_B)
	B = B - 40
	lock-S(A)
	read(A and Δ_A)
	A = A + 40
	lock-X(A and B)
	compare(Δ_A and Δ_B)
	write(A and Δ_A)
	write(B and Δ_B)
	release-X(A and B)

As we mentioned before, Prothos' concurrency control mechanism is less strict than the two-phase locking protocol, which leads to more transaction throughput. We now show an example to demonstrate this point.

The schedule in Table 5.14 is the modified version of the schedule in Table 5.9. The operations in this schedule follow the Prothos' concurrency control scheme we have just introduced.

In this schedule, Tr_2 can successfully commit whereas Tr_1 has to abort. When Tr_2 requests the exclusive locks on A and B, it will obtain the locks immediately even though A has already been shared-locked by Tr_1. When Tr_2 writes A back, it also increases the value of Δ_A so that the comparison of Δ_A and Δ_B in Tr_1 will fail, which forces Tr_1 to abort.

Table 5.14: A Schedule under Prothos' Concurrency Control Scheme

	Tr_1	Tr_2	
	lock-S(A)		
	read(A and Δ_A)		
	A = A - 20		
		lock-S(B)	
		read(B and Δ_B)	
		B = B - 40	
		lock-S(A)	
		read(A and Δ_A)	
		A = A + 40	
		lock-X(A and B)	will NOT wait
		compare(Δ_A and Δ_B)	
		write(A and Δ_A)	
		write(B and Δ_B)	
		release-X(A and B)	
		commits	
	lock-S(B)		
	read(B and Δ_B)		
	B = B + 20		
	lock-X(A and B)		
will fail since Δ_A has been changed by Tr_2	compare(Δ_A and Δ_B)		
	release-X(A and B)		
	aborts		

This schedule protects serializability. What's more, compared to the similar two-phase schedule in Table 5.9, this schedule does not generate the deadlock problem. The fact that Tr_2 successfully commits in this schedule demonstrates the assertion: Prothos' concurrency control scheme leads to more transaction throughput than the two-phase locking protocol does.

There are several reasons why Prothos uses this optimistic locking scheme to guard serializability and deadlock freedom. First of all, there might be enormous not-read-only transactions running concurrently. The exclusive locking time should be limited as much as possible so as for other transactions to be able to read data objects. The optimistic locking mechanism only issues the exclusive-mode locks when a transaction tries to enter the "commit" phase.

Moreover, the best scenario for the user is that there is no wait, and no rollback, either. However, in order to ensure both serializability and freedom from deadlock, one is inevitable. Since Prothos suggests short-duration transactions, minimum user-interaction transactions, and low-volume transactions, the current waiting and rollback schemes are satisfactorily valid.

There is a very important but implicit assumption in Prothos' optimistic locking scheme. That is, when a transaction enters into the "commit" phase, it would finish up in a certain period of time. In reality, the current POM module does not implement this assumption. As a result, there might be a **"starvation"** problem in Prothos which we will be investigating in Section 5.3.

5.2.4 Other Protocols

There are a lot more concurrency control schemes to protect serializability. All these do so by either delaying an operation or aborting the transaction that issued the operation. The most common ones are locking protocols, timestamp-ordering schemes, validation techniques, and multiversion schemes. We have already studied several locking protocols in great detail. In this section, we briefly review other schemes.

- **Timestamp-ordering schemes.** A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction Tr_i is smaller than the timestamp of transaction Tr_j, then the scheme ensures that the produced schedule is equivalent to a serial schedule in which Tr_i appears before Tr_j. This is done by rolling back a transaction whenever such an order is violated. More information about **timestamp-ordering schemes** can be found in [5], [36].

- **Validation techniques**. A validation scheme is an appropriate concurrency control method in cases where the majority of transactions are read-only transactions, and thus the rate of conflicts among these transactions may be very low. A unique fixed timestamp is associated with each transaction in the system. The serializability order is determined by the timestamp of the transaction. A transaction in this scheme is never delayed. It must, however, pass a validation test, in order to complete. If it does not pass the validation test, it is rolled back to its initial state. More information about **validation techniques** can be found in [3], [26].

- **Multiversion schemes.** A multiversion concurrency control scheme assumes that each write(O) operation creates a new version of data object O. When a read(O) operation is issued, the system selects one of the versions of O to be read. The concurrency control scheme ensures that the selection of the version to be read is done in a manner that ensures serializability. This is accomplished through the use of timestamps. A read operation always succeeds, while a write operation may result in the rollback of the transaction. More information about **multiversion schemes** can be found in [6], [41].

5.3 Model Checking Lock-Based Protocols

In Section 4.3.2, we addressed that Prothos' non-concurrent transaction model is too easily understandable to bother the model checking technique. The lock-based concurrency control protocols, on the other hand, are not straightforward to comprehend because even for a given number of transactions, there might exist a great deal of valid schedules under certain protocols. We could take full advantage of model checking to reason behavior of different kinds of protocols.

It is worthwhile to employ the model checking technique to understand those lock-based protocols since the concurrency control is probably the most important property of transaction management in Prothos.

5.3.1 Model Construction

In Section 5.2, we discussed two most significant features of a concurrency control scheme — serializability and deadlock freedom, and analyzed how different protocols ensure them. The models we build in this section would certainly center around these two key properties. Instead of arguing and exemplifying, we develop several formal models to demonstrate behavior of those protocols.

Figure 5.2 shows the state transition diagram of Prothos' optimistic locking concurrency control mechanism in the current implementation.

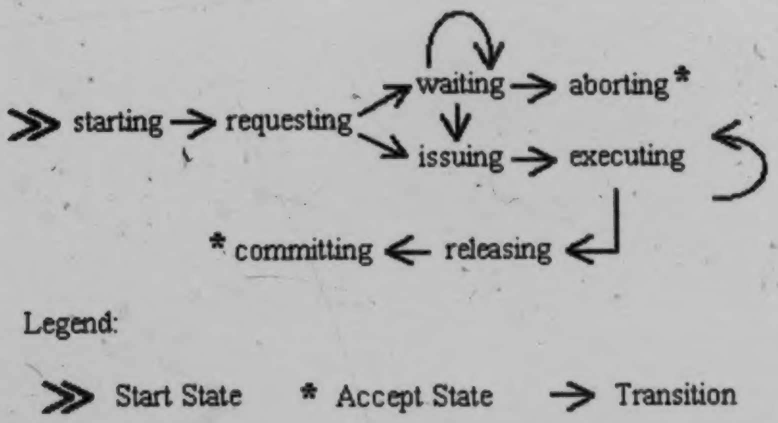

Figure 5.2: State Transition of Concurrency Control in Prothos (current implementation)

The state transition diagram in Figure 5.2 is captured from the current implementation of POM[1]. The code is the only "formalism" we can get.

Appendix A shows the SMV model built for checking the concurrency behavior of POM. The SMV file is self-explained, but here are some points we want to emphasize:

- The current implementation uses "per table locking", which means the locking object is a database table, not a specific data object. For each table, there is one "exclusive" transaction, one "waiting" transaction, and a set of "shared" transactions to record this table's locking status. In the SMV model, we use the variable "ca_w" and "ca_x" to represent the waiting transaction and exclusive-locking transaction, and use the array "ca_s" to represent the set of shared-locking transactions.

- The capitalized "Delta_A" represents the actual _Delta value in the database, so it is defined as a global variable in the SMV model. Each transaction keeps a local copy of "Delta_A" in "delta_A". When the transaction successfully committs, it increases the value of the global "Delta_A". The same rules can apply to "Delta_B".

- The global boolean variable "semaphore" in the SMV model is captured by the "/locked" directory creation in the implementation. If a transaction wants to obtain the exclusive lock on a table, it tries to create the "/locked" directory in the file system. Since the directory creation is mutually exclusive in UNIX/LINUX operating system, this guarantees that at most one process(transaction) can get the semaphore successfully and then enter into the critical section.

- The "waiting_timeout" defined in the SMV model is of type integer. This reflects the waiting timeout defined in the code. Currently, the waiting timeout is 20 seconds. So we use the range [0,20] as the scope of the "waiting_timeout" variable in the SMV model.

---

[1] Version 4.1.3.10, released on March 3, 2003. Author: James Konkin. © Copyright 2003 Avra Software Lab, Inc.

Given the above correspondences, we can focus on the SMV model to analyze and check the behavior of POM.

Every transaction starts in the "starting" state, and immediately goes to "requesting" state. If the transaction can get the locks for each data object, it enters to "issuing" state; otherwise, it gets into "waiting" state. When the transaction is waiting, a timeout is set for it. When the waiting timeout is expired, and the transaction still cannot get all the locks, it enters the "aborting" state, and the transaction is rolled back. If it can get all the locks during waiting, it will move to the "issuing" state.

The next state of "issuing" is "executing", in which the transaction does committing execution. It takes some time to finish the committing execution, so there is a transition loop defined in state "executing". When the executing is finished, the transaction releases all the locks, and commits. The "aborting" and "committing" states are accepting states in this model.

Note that finite-state limitations of the logic prevent us from building an SMV model that accepts arbitrary number of transactions. In the context of concurrency control protocols, it is rational to start reasoning with two interacted transactions. If the protocol is incorrect, problems would be disclosed in this two-transaction model for certain. If the protocol passes this test, from the architectural perspective, we could view these two transactions as a new "abstract" transaction. Since the "abstract" transaction holds serializability and freedom from deadlock inside and presents atomicity outside, it could be combined with another atomic transaction to form a new model. This new model would pass the test without doubt and be regarded as an "abstract" transaction again. In some cases, it might be possible to model check the inductive step to further verify the correctness of the protocol over arbitrary number of transactions.

5.3.2 Specifications

If we consider the two concurrent write-transaction situation, the example constructed in Section 5.1 is still suitable. The consistency constraint is that the sum, $A + B$, should always be \$300, which is used to check serializability.

For each transaction, if it eventually goes into "committing" or "aborting" state, it is not deadlocked. In another word, if a transaction makes progress, it is not deadlocked.

$$AG((A\_value\ +\ B\_value) = 300) \tag{5.1}$$

$$\begin{aligned}
&AG\ (\ (tr1.state = starting) \rightarrow \\
&(AF\ (tr1.state = committing \mid tr1.state = aborting)\ )\ ) \\
&\&\ AG(\ (tr2.state = starting) \rightarrow \\
&(AF\ (tr2.state = committing \mid tr2.state = aborting)\ )\ )
\end{aligned} \tag{5.2}$$

Spec 5.1 is the serializability specification and **Spec 5.2** is the deadlock specification. Both specifications are **false** in the general locking protocol model, which correctly validates that the non-restriction locking protocol could not guarantee either serializability or freedom from deadlock. The VW model checker[2] provides us a counter example every time the model fails to satisfy the specification.

The two-phase locking protocol model passes the serializability test with the value of **Spec 5.1** being **true**. **Spec 5.2** turns out to be **false** in this model, which confirms the announcement that the two-phase locking protocol may guide the system to a deadlock state.

We have gained some experience on writing specifications:

1. It is related to model construction.

   In our case, we cannot arbitrarily choose transaction's semantics since we want to check serializability. For example, the **Spec 5.1** does not allow us to define a transaction that withdraws \$35 from an account. The specifications indeed render the model construction in some sense.

2. Check whether the model itself is correct – $AG\ 0$.

---

[2] Author: Ken McMillan. © Copyright 2002 Cadence Design Systems, Inc.

Check firstly whether the model is correct by including the false specification ($AG\ 0$). If this specification is true, the model itself is wrong, and any other specification will be true. This model cannot check anything because it does not exist at all.

3. Make it false to check.

   Sometimes, it is necessary to write down "false" specifications. These specifications can perform as debuggers. They can help us find out what's wrong in the model. Exposing mistakes is also one of the advantages of formal methods [21].

4. Keep old "wrong" specifications for a later-version check.

   We find out that it is important to record the plausible specifications that were tried and rejected. These records will help improve the quality of specifications, guide future versions of the model, and prevent the repetition of unfruitful work.

5.3.3 Starvation

The SMV model we built based on Figure 5.2 satisfies **Spec 5.1**. But **Spec 5.2** turns out to be false in this case. This is not caused by deadlock though. The reason that **Spec 5.2** fails is the "starvation" problem as we mentioned in Section 5.2.3. Starvation is also known as the "indefinite postponement" situation.

One major assumption in Prothos' optimistic locking concurrency control scheme is that any transaction would come to an end when it went into the "commit" phase. If there were no such an assumption, the system would possibly get into the *starvation* state, that is, one process keeps seizing the execution thread and never halts; meanwhile, other processes are incapable of running because their desired resources are exclusively occupied by the "live" process. As a result, those transactions inevitably fall into an infinite loop of requesting, waiting, aborting, and requesting. This is apparently an undesirable situation since the system is not making progress at all.

In this case, **Spec 5.2** cannot be true. Although the transaction of being starved does go from "starting" to "aborting", the transaction that causes this starvation will stay in "executing" forever. Table 5.15 gives part of the counter example generated by the VW model checker. The trace shows why **Spec 5.2** is false in the model based on Figure 5.2.

Table 5.15: Part of the Counter Example

	running process	**state**
loop starts →	Tr_2	executing
	Tr_1	starting
	Tr_1	requesting
	Tr_1	waiting
	Tr_1	aborting
	Tr_1	starting
loop ends →	Tr_2	executing

So **Spec 5.2** not only checks freedom from deadlock, but also tests absence of starvation. To solve the starvation problem, we introduce another timeout[3] and build the future improvement model which is shown in Figure 5.3. When a transaction enters the "executing" state, the system automatically establishes a timeout for it. If the timeout expires and the transaction has not finished execution yet, this transaction is forced to roll back, and all locks on data objects currently held by it are released. The dollar sign in Figure 5.3 shows this new-added transition.

It is possible to set up such an executing timeout for transactions in Prothos to prevent the starvation problem. This could be a new feature for the new version of POM. We could configure an approximate running time for transactions since Prothos is designed for light-weight web applications. The SMV model built based on Figure 5.3 satisfies both **Spec 5.1** and **Spec 5.2**, which in effect verifies the correctness of the proposed solution.

---

[3] The first timeout set in the model is to guard deadlock freedom. If a transaction is waiting for another transaction to release the exclusive-mode lock on certain data object, a timeout is set up for it in order to avoid an arbitrarily long inactivity, see Section 5.2.3.

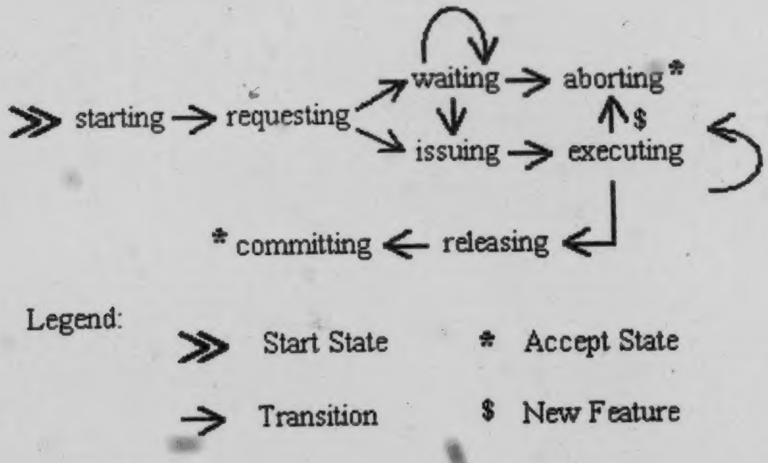
starting
requesting
waiting
aborting *
$
issuing
executing
* committing
releasing
Legend:
Start State
* Accept State
Transition
$ New Feature

Figure 5.3: State Transition of Concurrency Control in Prothos (future improvement)

5.3.4 Hermes

We have actually experienced an application built on top of Prothos that suffered from the starvation problem. In the fall 2002 academic term [23], a course project called "Hermes"[4] was developed using Prothos framework.

This project aims to deliver a database-style email client module for applications built on top of the Prothos framework. A user will be able to download his or her email messages from any standard account to be stored within the application's database. Multiple views of messages, including sorting, filtering and grouping, will be available to and configurable by the user to suit his or her workflow. The handling of attachments (sending, receiving and viewing) is key in any model of group workflow and will therefore be fully supported by the module.

It sometimes takes a long period of time for Hermes to retrieve email messages from different servers such as pop.mail.yahoo.com, imap.srv.ualberta.ca, and so forth. This long-duration transaction is possibly caused by the large number of email messages, the bulky size of attachments, or even the terrible traffic jam on the Internet.

---

[4] Authors: Tyler Botten, Matt Edwards, Francis Huber, Doug Hurst, Trina Magera, Ryan Warden, and Yifan Xu. © Copyright 2002 – 2003 Department of Computing Science, University of Alberta.

Since Prothos' POM does not take the starvation prevention into consideration, the transaction locks quite a few data objects and then moves into the "executing" phase. Due to several reasons, this transaction could take arbitrarily long and lead the system to the starvation state. Whenever this occurs, we have to manually remove the locks in the database in order for Hermes to resume running. This is obviously not the right way to use a framework. On one hand, such a "heavy-weight" application does not fit in Prothos' specific domain. On the other hand, the starvation defect discovered by model checking does affect the application.

We see from the above example that model checking helps present a critical observation. As a rule, the model checking technique plays a key role in analyzing the concurrency control protocols. If model checking (or some other formal methods) were used during Prothos' design phase, the starvation problem should have disappeared.

Although Wing [47] argued that formal methods can be applied in all phases of system development, as far as software architecture is concerned, we advise people apply formal methods to some vital parts in the software architecture **in the design phase**, or at least, earlier than the coding stage.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we have presented a formal way to understand the transaction management component's behavior in a software framework. The software component has been formally studied through three different layers. During this procedure, we argued that formal methods can indeed help improve software design and development. And as far as software architecture is concerned, they should be employed in the design phase of the whole software lifecycle.

We studied the software component's architecture first. Our methodology was to investigate several data models and then build up POM's ontology. The properties revealed in this level laid the groundwork for transaction-level behavior's discussion in the software architecture.

Then, we explored the atomicity of the single transaction model in the framework. The software component protected the database consistency both physically and logically in this second layer. The utilization of model checking in this context did not exhibit many exciting results.

The concurrency control scheme was of extreme importance for the transaction management component. Detailed descriptions of several lock-based protocols were provided. We took full advantage of model checking to illustrate and validate two most significant features of a concurrency control protocol — serializability and deadlock freedom. The starvation problem was discovered in the software framework by applying formal methods. We thus concluded our work in light of the question whether and how formalism can help software architecture.

The main contributions of this thesis are summarized as follows:

- Formalization draws out the "real" architecture. This is because by applying formal methods, we must study the system and pay attention to details. We demonstrate that "layering" is a good architectural principle.
- The goal of POM is to manipulate data objects as in object-oriented programs but they are actually kept around as in relational databases. The ontology provides the consistent terminology throughout the software framework and applications and among the stakeholders.
- As far as the single transaction model was concerned, we claimed that it was so easily understandable that the model checking technique did not fit in this context.
- Model checking played an important role in our work in checking the critical behavior of the software component: the original designs were correctly tested, and some problems and suggestions were pointed out.
- We put forward one potential improvement for the current version of Prothos. That is, in order to solve the starvation problem in the optimistic locking protocol, an approximate transaction finish timeout can be set.
- An SMV model was exploited in order to reason the key properties in the software framework. We offered some valuable recommendations on how to write specifications in SMV. We also inductively proved the correctness of our model from the architectural point of view.
- Our work guides the formal testing in the transaction management field in some sense. Serializability, deadlock freedom, and starvation freedom are necessary behaviors that need to be checked under a concurrency-supportive transaction management circumstance.

- "Formal methods" is still not the silver bullet. In order to understand, analyze, or reason a system, it is better to have a prototype, some sample code, or a concrete example.

Finally, we are able to answer the question — whether and how formalism can help software architecture. Based on this work, our answer is that, once again, formal methods can indeed help improve software design and development. And as far as software architecture is concerned, they should be employed in the design phase of the whole software lifecycle.

6.2 Future Work

While our work presents a complete formal way to comprehend the transaction management component's behavior in a software framework, there are still a number of areas for possible future research.

First of all, how to generate sufficient test case(s) is still an open issue. The test case depends on which feature in the architecture we want to check. For example, the current version of POM uses per table locking. For every table in the database, there is one "exclusive" transaction, one "waiting" transaction, and a set of "shared" transactions to record this table's locking status. In this case, two concurrent write-transaction case is not sufficient. In order to capture the locking implementation, we build a model of three write transactions and two read-only ones to test the specifications. This model turns out to be necessary, but still not sufficient. In our practice, we only employ base test cases. It will be beneficial to find out how to generate sufficient test case(s), given the "state space explosion" problem of model checking.

Moreover, it will be interesting and worthwhile to find out the appropriate formal methods to better illustrate the software component's properties in the first two levels — the basic database operational level and the non-concurrent transaction level. Our work mainly concentrates on the model checking technique, which we choose as the representative of formal methods. As we showed in Chapter 3 and 4, model checking is not the proper tool to some extent under these circumstances.

In addition, developing a collection of formal models of concurrency control protocols for the software component will greatly improve Prothos' adaptability. Besides lock-based protocols, as we introduced in Section 5.2.4, there are numerous schemes to guard serializability. Every protocol has its advantages and disadvantages. Model checking other schemes will soundly increase the value of the software component by making it easier to be adapted to other configurations.

Furthermore, there are several other inviting features in the concurrent transaction layer in Prothos that deserve further investigation, such as nested transactions, cascading rollback, etc. It will be absolutely meaningful to extend our models to check these behaviors. Since we pay most our attention to SMV models in particular, selecting other formal languages and tool sets will be another reasonable way to follow.

Last but not least, from the architectural standpoint, capturing the overall behavior of the entire software framework will definitely be necessary and attractive. In this thesis, we have explored the applicability of SMV models in analyzing the transaction management component's behavior. Making good use of formal methods across a range of components will be of benefit to ultimately comprehend the behavior of the whole framework. Without a doubt, this kind of work will shed further light on how to better answer the question — ***whether and how formalism can help software architecture.***

Bibliography

[1] S. Alagic. *Relational Database Technology.* Springer-Verlag New York, Inc., 1986.

[2] R. J. Allen. A formal approach to software architecture. Ph.D. Thesis, Technical Report Number: CMU-CS-97-144, Carnegie Mellon University, 1997.

[3] M. Bassiouni. Single-site and distributed optimistic protocols for concurrency control. In *IEEE Transactions on Software Engineering, Volume SE-14, Number 8 (August 1988), pages 1071–1080*, 1988.

[4] D. Begay. The eurotri case, some aspects of using formal methods in an industrial project. Internal Report, Techinical Report Number: 1141-96, LaBRI, 1996.

[5] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proceedings of the International Conference on Very Large Data Bases (1980), pages 285–300*, 1980.

[6] P. A. Bernstein, N. Goodman, and M. Y. Lai. Analyzing concurrency control when user and system operations differ. In *IEEE Transactions on Software Engineering, Volume SE-9, Number 3 (May 1983), pages 233–239*, 1983.

[7] A. Bouhoula. Automated theorem proving by test set induction. In *Journal of Symbolic Computation, 23(1): pages 47–77*, 1997.

[8] J. P. Burgess. Basic tense logic. In *Handbook of Philosophical Logic. Volume 2: Extensions of Classical Logic, pages 89–134*, 1984.

[9] R. W. Butler, V. A. Carreno, B. L. Di Vito, K. J. Hayhurst, and P. S. Miner. Nasa langley's research and technology-transfer program in formal methods. In *Proceedings of the Tenth Annual Conference on Computer Assurance (1995), pages 135–150*, 1995.

[10] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, volume 131 of Lecture Notes in Computer Science, Springer-Verlag*, 1981.

[11] F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. In *IEEE Transactions on Software Engineering, Volume SE-2, Number 2 (June 1976), pages 80–86*, 1976.

[12] A. Descartes and T. Bunce. *Programming the Perl DBI.* O'Reilly & Associates, Inc., 2000.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. In *Communications of the ACM, Volume 19, Number 11 (November 1976), pages 624–633*, 1976.

[14] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the International Conference on Software Engineering (1997), pages 491–501*, 1997.

[15] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. *Designing Object-Oriented Frameworks.* CRC Handbook of Object Technology, CRC Press, 1998.

[16] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of ACM SIGSOFT (1994): Foundations of Software Engineering, pages 175–188*, 1994.

[17] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON (1997), pages 169–183*, 1997.

[18] R. Greenlaw and H. J. Hoover. *Fundamentals of the Theory of Computation, Principles and Practice.* Morgan Kaufmann Publishers, Inc., 1998.

[19] Object Management Group. The common object request broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

[20] T. R. Gruber. A translation approach to portable ontology specifications. In *Knowledge Acquisition, Volume 5, Number 2, pages 199–220*, 1993.

[21] A. Hall. Seven myths of formal methods. In *IEEE Software, Volume 6, Number 9 (September 1990), pages 11–19*, 1990.

[22] R. Hayes and R. D. Schlichting. Facilitating mixed language programming in distributed systems. In *IEEE Transactions on Software Engineering, Volume SE-13, Number 12 (December 1987), pages 54–64*, 1987.

[23] H. J. Hoover. Cmput 401: Software process and product management. In *Department of Computing Science, University of Alberta*, Fall term, 2002.

[24] D. Hou and H. J. Hoover. Towards specifying constraints for object-oriented frameworks. In *Journal of Information Systems Frontiers, 4(4): pages 393–407*, 2002.

[25] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, Inc., 1991.

[26] H. T. Kung and J. T. Robinson. Optimistic concurrency control. In *ACM Transactions on Database Systems, Volume 6, Number 2 (June 1981), pages 312–326*, 1981.

[27] D. A. Lamb. Idl: Sharing intermediate representations. In *ACM Transactions on Programming Languages and Systems, Volume 9, Number 3 (July 1987), pages 297–318*, 1987.

[28] D. C. Luckham and J. Vera. An event-based architecture definition language. In *IEEE Transactions on Software Engineering, Volume SE-21, Number 9 (September 1995), pages 717–734*, 1995.

[29] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of ACM SIGSOFT (1996): Sourth Symposium Foundations of Software Engineering, pages 3–14*, 1996.

[30] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[31] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering, Volume SE-26, Number 1 (January 2000), pages 70–93*, 2000.

[32] T. H. Merrett. *Relational Information Systems*. Reston Publishing Company, Inc., 1984.

[33] C. J. Nix and B. P. Collins. The use of software engineering, including the z notation, in the development of cics. In *Quality Assurance, September 1988: pages 103–110*, 1988.

[34] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software Engineering Notes, 17(4): pages 40–52*, 1992.

[35] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. In *Journal of Systems and Software, 6(4): pages 307–334*, 1986.

[36] D. Reed. Implementing atomic actions on decentralized data. In *ACM Transactions on Computer Systems, Volume 1, Number 1 (February 1983), pages 3–23*, 1983.

[37] Microsoft Research. Software model checking project at microsoft research. http://research.microsoft.com/slam.

[38] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[39] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. In *IEEE Transactions on Software Engineering, Volume SE-21, Number 4 (April 1995), pages 314–335*, 1995.

[40] M. Shaw and D. Garlan. *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, Inc., 1996.

[41] A. Silberschatz. A multi-version concurrency control scheme with no rollbacks. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (1982), pages 216–223*, 1982.

[42] Avar Software. Prothos reference document on peoria. http://peoria.cs.ualberta.ca/~fraser/prothos/prothos.htm.

[43] D. Szafron. Cmput 625: Object-oriented programming languages. In *Department of Computing Science, University of Alberta*, Fall term, 2002.

[44] W. F. Tichy. Software development control based on module interconnection. In *Proceedings of the International Conference on Software Engineering (1979), pages 29–41*, 1979.

[45] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Inc., 1982.

[46] P. Valduriez and G. Gardarin. *Analysis and Comparison of Relational Database Systems.* Springer-Verlag New York, Inc., 1989.

[47] J. M. Wing. A specifier's introduction to formal methods. In *IEEE Computer, Volume 23, Number 9 (September 1990), pages 8–24*, 1990.

Appendix A

The SMV Model

```
----------------------------------------------------------------
----------------------------------------------------------------
-- Subject: Model checking the locking schema in Prothos' POM (v 4.1.3.10)
-- Note: Give semantics to each transaction to check serializability
--       deadlock freedom, and starvation freedom
-- Author: Nan Niu (nan@cs.ualberta.ca)
-- Create Date: 17:46, April 24, 2003
-- Last Modified Date: 05:30, Aug 08, 2003
-- Last Modified By: Nan Niu (nan@cs.ualberta.ca)
----------------------------------------------------------------
----------------------------------------------------------------

-- Model checking the locking schema in Prothos, Version 4.1.3.10
-- released date: March 5, 2003
-- Code mainly focused on DB.pm
-- In this model, there are two transactions: tr1, tr2
-- Two DB tables, and two DB records accordingly: A and B
-- with initial value of 100 and 200 respectively
-- As we're using per table locking
-- And A and B are the only record considered in each table
-- We use A_s, A_w, and A_x instead of table names
-- The relationship is:

-- tr1: transferring 20 from A to B
-- tr2: transferring 40 from B to A

-- Note: the 'w' (waiting) lock is set automatically;
-- No transaction requests a lock mode 'w' at all.

MODULE main

VAR

    -- GLOBAL variables defined here
    -- the directory '/locked' is the semaphore in the locking schema
    semaphore : boolean;
```

```
      - - the capitalized 'Delta_A' and 'Delta_B' are actual values in the
database
      - - since there are only two write-transactions, the maximum value of the
global delta will be 2
      Delta_A : 0..2;
      Delta_B : 0..2;

      A_value : 0..500;
      B_value : 0..500;

      A_s : array 1..2 of {tr1, tr2, nil};
      A_w : {tr1, tr2, nil};
      A_x : {tr1, tr2, nil};

      B_s : array 1..2 of {tr1, tr2, nil};
      B_w : {tr1, tr2, nil};
      B_x : {tr1, tr2, nil};

      - - two transactions defined as follows
      - - the instance variable like 'state'
      - - 'local_A' are defined inside the module

      tran1 : process transaction(tr1, semaphore, Delta_A, Delta_B, A_value,
B_value, A_s, A_w, A_x, B_s, B_w, B_x);
      tran2 : process transaction(tr2, semaphore, Delta_A, Delta_B, A_value,
B_value, A_s, A_w, A_x, B_s, B_w, B_x);


ASSIGN

      init(semaphore) := 0;

      init(Delta_A) := 0;
      init(Delta_B) := 0;

      init(A_value) := 100;
      init(B_value) := 200;

      init(A_s[1]) := nil;
      init(A_s[2]) := nil;
      init(A_w) := nil;
      init(A_x) := nil;

      init(B_s[1]) := nil;
      init(B_s[2]) := nil;
      init(B_w) := nil;
      init(B_x) := nil;

- - Specifications

- - check whether the model itself is correct
```

```
SPEC
    AG 0

- - check semaphore

SPEC
    AG ( (tran1.state=issuing | tran1.state=executing |
tran1.state=releasing) → semaphore )


SPEC
    AG ( (tran2.state=issuing | tran2.state=executing |
tran2.state=releasing) → semaphore )


- - check the waiting_timeout

SPEC
    AG ( (tran2.state=requesting) → (tran2.waiting_timeout=0) )

SPEC
    AG ( (tran1.state=waiting) → !(tran1.waiting_timeout=0) )


- - check serializability

SPEC
    AG (A_value + B_value = 300)


- - check deadlock freedom and starvation freedom

SPEC
    AG ( (tran1.state=starting) → AF (tran1.state=committing |
tran1.state=aborting) )

SPEC
    AG ( (tran2.state=starting) → AF (tran2.state=committing |
tran2.state=aborting) )


- - make the specification "false" to give us a counter example

SPEC
    AG !(tran1.waiting_timeout=17)


- - A transaction has 8 state: starting, requesting,
- - waiting, issuing, aborting, executing, releasing, committing

- - each transaction will keep a local copy of delta_A and delta_B
- - in order to perform the optimistic checking
- - about the waiting timeout
- - a waiting timeout is 20 means the transaction starts waiting;
- - when the timeout is decreased to 1, it stops waiting, and rolls back;
- - and if the value is 0, the transaction is not waiting at all

- - about the executing timeout
- - an executing timeout is 10 means the transaction starts executing;
- - when the timeout is decreased to 1, it stops executing,
- - and goes to aborting;
- - and if the value if 0, the transaction is not executing at all


MODULE transaction(id, semaphore, A_value, B_value, A_s, A_w, A_x, B_s,
B_w, B_x)


VAR
    state : {starting, requesting, waiting, issuing, aborting, executing,
aborting, committing};

    delta_A : 0..2;
    delta_B : 0..2;

    waiting_timeout : 0..20;

    executing_timeout : 0..10;


ASSIGN
    init(state) := starting;
    init(delta_A) := 0;
    init(delta_B) := 0;
    init(waiting_timeout) := 0;
    init(executing_timeout) := 0;


    - - The following annotations are important
    - - They give the reader the basic idea of model construction


    - - TRANSITION 001 : starting → requesting

    - - CASE 001 (TRANSITION 001)
    - - affects: state only


    - - TRANSITION 002 : requesting → issuing

    - - CASE 002 (TRANSITION 002)
    - - you can get the semaphore, both A_x and B_x are 'nil'
    - - A_w is either 'nil' or 'id'; so is B_w
```

```
-- affects: state, semaphore, A_x, B_x

-- TRANSITION 003 : requesting → waiting

-- CASE 003 (TRANSITION 003)
-- you're in state 'requesting', but cannot get semaphore
-- only check whether you should set A_w and B_w or not
-- affects: state, waiting_timeout, A_w, B_w

-- CASE 004 (TRANSITION 003)
-- you're in state 'requesting', and you can get the semaphore
-- however, you just cannot get the lock you want
-- affects: state, waiting_timeout, A_w, B_w


-- TRANSITION 004 : waiting → issuing

-- CASE 005 (TRANSITION 004)
-- pretty much like CASE 002
-- you're in the state 'waiting', and you get the semaphore
-- every locking condition is good for you
-- affects: state, waiting_timeout, semaphore, A_w, A_x, B_w, B_x


-- TRANSITION 005 : waiting → waiting

-- CASE 006 (TRANSITION 005)
-- pretty much like CASE 003
-- you're in state 'waiting', but cannot get semaphore
-- at the same time, your waiting_timeout is NOT 1,
-- and is NOT 0, either
-- should check A_w and B_w
-- affects: state, waiting_timeout, A_w, B_w

-- CASE 007 (TRANSITION 005)
-- pretty much like CASE 004
-- you're in state 'waiting', and you can get the semaphore
-- at the same time, your waiting_timeout is NOT 1,
-- and is NOT 0, either
-- however, you just cannot get the lock you want
-- affects: state, waiting_timeout, A_w, B_w


-- TRANSITION 006 : waiting → aborting

-- CASE 008 (TRANSITION 006)
-- pretty much like CASE 006
-- you're in state waiting', but cannot get semaphore
-- at the same time, your waiting_timeout is 1, don't wait any more
-- go to 'aborting'
-- should check A_w and B_w
-- affects: state, waiting_timeout, A_w, B_w

-- CASE 009 (TRANSITION 006)
```

```
- - pretty much like CASE 007
- - you're in state 'waiting', and you can get the semaphore
- - however, you cannot get the lock you want
- - at the same time, your waiting_timeout is 1, don't wait any more
- - go to 'aborting'
- - affects: state, waiting_timeout, A_w, B_w


- - TRANSITION 007 : issuing → executing

- - CASE 010 (TRANSITION 007)
- - the new feature in POM – executing timeout
- - affects: state, executing_timeout


- - TRANSITION 008 : executing → aborting

- - CASE 011 (TRANSITION 008)
- - you're in state 'executing',
- - and the executing_timeout is 1, don't execute any more,
- - go to 'aborting'
- - affects: state, semaphore, A_x, B_x, executing_timeout


- - TRANSITION 009 : executing → executing
- - TRANSITION 010 : executing → releasing

- - CASE 012 (TRANSITION 009 & 010)
- - you're in state 'executing',
- - you can go either 'executing' or 'releasing' arbitrarily
- - given CASE 011 defines first, that is, the executing_timeout is not 1
- - affects: state

- - CASE 013 (TRANSITION 009)
- - you're in state 'executing',
- - you have to check the executing_timeout
- - that is, if the executing_timeout is not 1 or not 0,
- - decreasing the timeout
- - affects: executing_timeout

- - CASE 014 (TRANSITION 010)
- - you're in state 'releasing',
- - you have to make sure the executing_timeout is 0
- - affects: executing_timeout


- - TRANSITION 011 : releasing → committing

- - CASE 015 (TRANSITION 011)
- - you're in state 'releasing', you must go to 'committing' next
- - affects: state, semaphore, A_value, B_value, A_x, B_x


next(state) := case
```

```
        - - CASE 001
        state=starting : requesting;

        - - CASE 002
        state=requesting & !semaphore & (A_x=nil & B_x=nil) & (
(A_w=nil | A_w=id) & (B_w=nil | B_w=id) ) : issuing;

        - - CASE 003
        state=requesting & semaphore : waiting;

        - - CASE 004
        state=requesting & !semaphore & ( !(A_x=nil) | !(B_x=nil) ) :
waiting;
        state=requesting & !semaphore & (A_x=nil & !(A_w=nil) &
!(A_w=id) ) : waiting;
        state=requesting & !semaphore & (B_x=nil & !(B_w=nil) &
!(B_w=id) ) : waiting;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : issuing;

        - - CASE 006
        state=waiting & semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) : waiting;

        - - CASE 007
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & ( !(A_x=nil) | !(B_x=nil) ) : waiting;
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & (A_x=nil & !(A_w=nil) & !(A_w=id) ) : waiting;
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & (B_x=nil & !(B_w=nil) & !(B_w=id) ) : waiting;

        - - CASE 008
        state=waiting & semaphore & waiting_timeout=1 : aborting;

        - - CASE 009
        state=waiting & !semaphore & waiting_timeout=1 & ( !(A_x=nil) |
!(B_x=nil) ) : aborting;
        state=waiting & !semaphore & waiting_timeout=1 & (A_x=nil &
!(A_w=nil) & !(A_w=id) ) : aborting;
        state=waiting & !semaphore & waiting_timeout=1 & (B_x=nil &
!(B_w=nil) & !(B_w=id) ) : aborting;

        - - CASE 010
        state=issuing : executing;

        - - CASE 011
        state=executing & executing_timeout=1 : aborting;

        - - CASE 012
```

```
          state=executing : {executing, releasing};

          - - CASE 015
          state=releasing & (delta_A=Delta_A & delta_B=Delta_B) :
committing;

          - - otherwise, stay unchanged
          1 : state;

     esac;


     next(delta_A) := case

          - - CASE 015
          state=releasing & (delta_A=Delta_A & delta_B=Delta_B) : delta_A
+ 1;
     esac;


     next(delta_B) := case

          - - CASE 015
          state=releasing & (delta_A=Delta_A & delta_B=Delta_B) : delta_B
+ 1;
     esac;


     next(waiting_timeout) := case

          - - CASE 003
          state=requesting & semaphore : 20;

          - - CASE 004
          state=requesting & !semaphore & ( !(A_x=nil) | !(B_x=nil) ) : 20;
          state=requesting & !semaphore & (A_x=nil & !(A_w=nil) &
!(A_w=id) ) : 20;
          state=requesting & !semaphore & (B_x=nil & !(B_w=nil) &
!(B_w=id) ) : 20;

          - - CASE 005
          state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : 0;

          - - CASE 006
          state=waiting & semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) : waiting_timeout - 1;

          - - CASE 007
          state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & ( !(A_x=nil) | !(B_x=nil) ) : waiting_timeout - 1;
```

```
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & (A_x=nil & !(A_w=nil) & !(A_w=id) ) :
waiting_timeout - 1;
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & (B_x=nil & !(B_w=nil) & !(B_w=id) ) :
waiting_timeout - 1;

        - - CASE 008
        state=waiting & semaphore & waiting_timeout=1 : 0;

        - - CASE 009
        state=waiting & !semaphore & waiting_timeout=1 & ( !(A_x=nil) |
!(B_x=nil) ) : 0;
        state=waiting & !semaphore & waiting_timeout=1 & (A_x=nil &
!(A_w=nil) & !(A_w=id) ) : 0;
        state=waiting & !semaphore & waiting_timeout=1 & (B_x=nil &
!(B_w=nil) & !(B_w=id) ) : 0;

        - - otherwise, stay unchanged
        1 : waiting_timeout;

    esac;


    next(executing_timeout) := case

        - - CASE 010
        state=issuing : 10;

        - - CASE 011
        state=executing & executing_timeout=1 : 0;

        - - CASE 013
        state=executing & !(executing_timeout=1) &
!(executing_timeout=0) : executing_timeout - 1;

        - - CASE 014
        state=releasing : 0;
        - - otherwise, stay unchanged
        1 : executing_timeout;

    esac;


    next(semaphore) := case

        - - CASE 002
        state=requesting & !semaphore & (A_x=nil & B_x=nil) & (
(A_w=nil | A_w=id) & (B_w=nil | B_w=id) ) : 1;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : 1;

        - - CASE 011
        state=executing & executing_timeout=1 : 0;

        - - CASE 015
        state=releasing : 0;

        - - otherwise, stay unchanged
        1 : semaphore;

    esac;


    next(A_value) := case

        - - avoid A_value out of range
        A_value > 460 : A_value;
        A_value < 20 : A_value;

        - - CASE 011
        state=releasing & id=tr1 : A_value - 20;
        state=releasing & id=tr2 : A_value + 40;

        - - otherwise, stay unchanged
        1 : A_value;

    esac;


    next(B_value) := case

        - - avoid B_value out of range
        B_value > 480 : B_value;
        B_value < 40 : B_value;

        - - CASE 011
        state=releasing & id=tr1 : B_value + 20;
        state=releasing & id=tr2 : B_value - 40;

        - - otherwise, stay unchanged
        1 : B_value;

    esac;


    next(A_s[1]) := case

        - - otherwise, stay unchanged
        1 : A_s[1];
    esac;


    next(A_s[2]) := case

         - - otherwise, stay unchanged
         1 : A_s[2];

    esac;


    next(A_w) := case

         - - CASE 002
         state=requesting & !semaphore & (A_x=nil & B_x=nil) & A_w=id :
nil;

         - - CASE 003
         state=requesting & semaphore & A_w=nil : id;

         - - CASE 004
         state=requesting & !semaphore & ( !(A_x=nil) & A_w=nil ) : id;

         - - CASE 005
         state=waiting & !semaphore & (A_x=nil & B_x=nil) & A_w=id :
nil;

         - - CASE 006
         state=waiting & semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & A_w=nil : id;

         - - CASE 007
         state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & ( !(A_x=nil) & A_w=id ) : nil;

         - - CASE 008
         state=waiting & semaphore & waiting_timeout=1 & A_w=id : nil;

         - - CASE 009
         state=waiting & !semaphore & waiting_timeout=1 & A_w=id : nil;

         - - otherwise, stay unchanged
         1 : A_w;

    esac;


    next(A_x) := case

         - - CASE 002
         state=requesting & !semaphore & (A_x=nil & B_x=nil) & (
(A_w=nil | A_w=id) & (B_w=nil | B_w=id) ) : id;
```

```
        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : id;

        - - CASE 011
        state=executing & executing_timeout=1 & A_x=id : nil;

        - - CASE 015
        state=releasing & (A_x=id | A_x=nil) : nil;

        - - otherwise, stay unchanged
        1 : A_x;

    esac;


    next(B_s[1]) := case

        - - otherwise, stay unchanged
        1 : B_s[1];

    esac;


    next(B_s[2]) := case

        - - otherwise, stay unchanged
        1 : B_s[2];

    esac;


    next(B_w) := case

        - - CASE 002
        state=requesting & !semaphore & (A_x=nil & B_x=nil) & B_w=id :
nil;

        - - CASE 003
        state=requesting & semaphore & B_w=nil : id;

        - - CASE 004
        state=requesting & !semaphore & ( !(B_x=nil) & B_w=nil ) : id;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & B_w=id :
nil;

        - - CASE 006
        state=waiting & semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & B_w=nil : id;
```

```
        - - CASE 007
        state=waiting & !semaphore & ( !(waiting_timeout=1) &
!(waiting_timeout=0) ) & ( !(B_x=nil) & B_w=id ) : nil;

        - - CASE 008
        state=waiting & semaphore & waiting_timeout=1 & B_w=id : nil;

        - - CASE 009
        state=waiting & !semaphore & waiting_timeout=1 & B_w=id : nil;

        - - otherwise, stay unchanged
        1 : B_w;

    esac;


    next(B_x) := case

        - - CASE 002
        state=requesting & !semaphore & (A_x=nil & B_x=nil) & (
(A_w=nil | A_w=id) & (B_w=nil | B_w=id) ) : id;

        - - CASE 005
        state=waiting & !semaphore & (A_x=nil & B_x=nil) & ( (A_w=nil |
A_w=id) & (B_w=nil | B_w=id) ) : id;

        - - CASE 011
        state=executing & executing_timeout=1 & B_x=id : nil;

        - - CASE 015
        state=releasing & (A_x=id | A_x=nil) : nil;

        - - otherwise, stay unchanged
        1 : B_x;

    esac;


FAIRNESS
    running
```